Aspects of p-adic computation

Christopher James Doris

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of Doctor of Philosophy in the Faculty of Science.

June 2019

Word count: 43,260

Abstract

We present a collection of new algorithms and approaches to several aspects of p-adic computation including:

- computing the Galois group of a polynomial defined over a p-adic field;
- computing the conductor of a 2-adic hyperelliptic curve of genus 2;
- representing p-adic numbers exactly using lazy arithmetic; and
- finding the roots of a system of polynomials in several variables over a p-adic field.

In all cases, these algorithms are new or improve significantly on the previous state of the art. Most are implemented in the Magma computer algebra system, with source code freely available on the author's website.

We have used these to prove the conductors of all genus 2 curves in the L-functions and modular forms database (LMFDB), which were previously conjectural, and have verified the Galois groups in the local fields database. We have also produced tables of previously unknown Galois groups, also available on the author's website.

Dedication and Acknowledgements

Dedicated to Alison Rowley
for putting up with this for four years.

Many thanks to my supervisor Tim Dokchitser, who is skillful in both listening and explaining. Our weekly meetings were invaluable.

Thanks also to those who have provided interesting test cases for our algorithms, or who have tested and found bugs in our code: Lassina Dembele, John Jones, Ciaran Schembri and Haluk Sengun.

Thanks also to my examiners Jürgen Klüners and Xavier Caruso for their careful reading of this manuscript and helpful suggestions.

Finally, thanks to EPSRC and GCHQ for generously funding this research.

Author's Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ..... DATE:.....

Contents

List of Figures

List of Tables

Chapter I

Introduction

Since their introduction in 1897 by Hensel [36], the p-adic numbers have become a fundamental tool in number theory. Eighty years later we see the rise in popularity of computer algebra systems such as Maxima and Maple and, more recently, Magma, Mathematica and SageMath. Of these, Magma and SageMath include highly featured representations of p-adic numbers, fields and extensions, enabling a wide range of p-adic computations.

In this thesis, we present a collection of new algorithms and approaches to several aspects of p-adic computation. Each chapter deals with a separate aspect and can be read independently of the others. Where necessary, a chapter may begin with a foreword stating whether it has been published already and properly attributing shared work.

In Chapter II we present a family of algorithms for computing the Galois group of a polynomial defined over a p-adic field. Apart from the "naive" algorithm, these are the first general algorithms for this task. As an application, we compute the Galois groups of all totally ramified extensions of $\mathbb{Q}_2$ of degrees 18, 20 and 22, tables of which are available online.

In Chapter III we give a brief re-exposition of the theory due to Pauli and Sinclair of ramification polygons of Eisenstein polynomials over p-adic fields, their associated residual polynomials and an algorithm to produce all extensions for a given ramification polygon. We supplement this with an algorithm to produce all ramification polygons of a given degree, and hence we can produce all totally

ramified extensions of a given degree.

In Chapter IV we describe two new packages `ExactpAdics` and `ExactpAdics2` for the Magma computer algebra system for working with p-adic numbers exactly, in the sense that numbers are represented lazily to infinite p-adic precision. This has the benefits of increasing user-friendliness and speeding up some computations, as well as forcibly producing provable results. The two packages use different methods for lazy evaluation, which we describe and compare in detail. The intention is that this article will be of benefit to anyone wanting to implement similar functionality in other languages.

In Chapter V we give an algorithm to compute the conductor for curves of genus 2. It is based on the analysis of 3-torsion of the Jacobian for genus 2 curves over 2-adic fields. We verify that the previously conjectural conductors in the LMFDB are correct.

In Chapter VI we describe a new algorithm for finding the roots of a system of n polynomials in n variables over a p-adic field. It is a generalization to multivariate polynomials of an "OM algorithm" for univariate factorization specialized to root-finding.

We assume the reader is familiar with the fundamentals of p-adic fields, such as [60, Ch. I–V] or [46, Ch. II].

Non-Galois ramification theory and its relation to ramification polygons is given an overview in Chapter IV §9.6. For a more in-depth discussion see [35] and [46, Ch. III].

Chapter II

Galois Groups

1 Introduction

In this article we consider the following problem, the p-adic instance of the forward Galois problem: given a p-adic field K and a polynomial $F(x) \in K[x]$ over that field, what is its Galois group $G := \mathrm{Gal}(F/K)$?

Over any field for which polynomial factorization algorithms are known, the forward Galois problem can always be solved with the **naive algorithm**: explicitly compute the splitting field of F by repeatedly adjoining a root of it to the base field, and then explicitly compute the automorphisms of the splitting field. To date, there is no general solution to the p-adic forward Galois problem other than the naive algorithm.

This article presents a general algorithm. In practice, it can for example quickly determine the Galois group of most irreducible polynomials of degree 16 over $\mathbb{Q}_2$ and has been used to compute some non-trivial Galois groups at degree 32. It has been tested on polynomials defining all extensions of $\mathbb{Q}_2$, $\mathbb{Q}_3$ and $\mathbb{Q}_5$ of degree up to 12, all extensions of $\mathbb{Q}_2$ of degree 14, and all totally ramified extensions of $\mathbb{Q}_2$ of degrees 18, 20 and 22, the latter three being new. See §13.

Our implementation is publicly available [27] and pre-computed tables of Galois groups are available from here also.

1.1 Overview of algorithm

Our algorithm uses the "resolvent method". We now describe a concrete instance.

Suppose $F(x) \in \mathbb{Q}_p[x]$ is irreducible of degree d, and therefore defines an extension $L/\mathbb{Q}_p$ of degree d.

The ramification filtration of this extension is a tower $L_t = L/\ldots/L_0 = \mathbb{Q}_p$. Let $F_1(x) \in \mathbb{Q}_p[x]$ be a defining polynomial for $L_1/\mathbb{Q}_p$. By Krasner's lemma, any polynomial in $\mathbb{Q}[x]$ sufficiently close to F_1 is also a defining polynomial, so we may take $F_1 \in \mathbb{Q}[x]$. It is irreducible and so defines the number field $\mathcal{L}_1/\mathcal{L}_0 = \mathbb{Q}$ which has a unique completion embedding into L_1. Repeating this procedure up the tower, we obtain the tower of number fields $\mathcal{L} = \mathcal{L}_t/\ldots/\mathcal{L}_0 = \mathbb{Q}$ such that $\mathcal{L}$ embeds uniquely into L. We call $\mathcal{L}/\mathbb{Q}$ a **global model** of $L/\mathbb{Q}_p$.

Let $d_i := (\mathcal{L}_i : \mathcal{L}_{i-1}) = (L_i : L_{i-1})$, then $\mathrm{Gal}(\mathcal{L}_i/\mathcal{L}_{i-1}) \leq S_{d_i}$ and therefore $\mathrm{Gal}(\mathcal{L}/\mathbb{Q}) \leq W := S_{d_t} \wr \cdots \wr S_{d_1}$. Observe also that naturally $\mathrm{Gal}(L/\mathbb{Q}_p) \leq \mathrm{Gal}(\mathcal{L}/\mathbb{Q})$ since the left hand side is a decomposition group of the right hand side.

Suppose $\alpha_1 \in \mathcal{L}$ generates $\mathcal{L}/\mathbb{Q}$, and let $\alpha_2, \ldots, \alpha_d \in \bar{\mathbb{Q}}$ be its $\mathbb{Q}$-conjugates. Suppose we choose some subgroup $U \leq W$, find an **invariant** $I \in \mathbb{Z}[x_1, \ldots, x_d]$ such that $\mathrm{Stab}_W(I) = U$ and compute the **resolvent**

$$R(x) = \prod_{wU \in W/U} (t - wU(I)(\alpha_1, \ldots, \alpha_d)) \in \mathbb{Z}[t]$$

by finding sufficiently precise complex approximations to $\alpha_1, \ldots, \alpha_d$, giving a complex approximation to R, whose coefficients we can then round to $\mathbb{Z}$.

One can show that $\mathrm{Gal}(R/\mathbb{Q}) = q(\mathrm{Gal}(\mathcal{L}/\mathbb{Q}))$ and hence $\mathrm{Gal}(R/\mathbb{Q}_p) = q(\mathrm{Gal}(L/\mathbb{Q}_p)) = q(\mathrm{Gal}(F/\mathbb{Q}_p))$ where $q : W \to S_{W/U}$ is the action of W on the cosets of U.

In particular, if we define $s(G)$ to be the multiset of the sizes of orbits of the permutation group G, and we let S be the multiset of the degrees of the factors of R over K, then $s(q(\mathrm{Gal}(F/\mathbb{Q}_p))) = S$.

We compute the set $\mathcal{G}$ of all transitive subgroups of W, so that $\mathrm{Gal}(F/\mathbb{Q}_p) \in \mathcal{G}$. If $|\mathcal{G}| > 1$, we search through the subgroups $U \leq W$ in index order until we find one such that $\{s(q(G)) : G \in \mathcal{G}\}$ contains at least two elements. We then compute the corresponding resolvent $R(t) \in \mathbb{Z}[t]$, factorize it over $\mathbb{Q}_p$ and let S be the multiset

of degrees of factors, and replace $\mathcal{G}$ by $\{G \in \mathcal{G} : s(q(G)) = S\}$. Observe that $\mathcal{G}$ is now strictly smaller than it was before, and we still have $\mathrm{Gal}(F/\mathbb{Q}_p) \in \mathcal{G}$.

We repeat this process until $|\mathcal{G}| = 1$, at which point this single group is the Galois group and we are done.

In §2.4 we describe our precise formulation of this algorithm.

We have described one method of producing a global model, which results in the group W (relative to which we compute resolvents) being a wreath product of symmetric groups. It is better for W to be as small as possible, since this will reduce the index $(W : U)$ required, and hence also reduce $\deg R$. In §4 we discuss some other constructions. The best constructions take advantage of the simple structure of the Galois group of a "singly ramified" extension, something like C_d for unramified extensions, $C_d \rtimes (\mathbb{Z}/d\mathbb{Z})^\times$ for tame extensions and $C_p^k \rtimes H$ for wild extensions. We can also produce global models for reducible F using global models for its factors.

In this example, we deduced the Galois group by enumerating the set $\mathcal{G}$ of all possibilities and then eliminating candidates. This is the "group theory" part of the algorithm. We have other methods which avoid enumerating all subgroups of W, and instead work down the graph of subgroups of W. These are discussed in §5.

The function s taking a group and returning the multiset of sizes of its orbits is a "statistic", and there are other choices. These are discussed in §6. Some statistics provide more information than others, and therefore can result in smaller indices $(W : U)$ being required, but this comes at the expense of taking longer to compute.

We search for U by enumerating all the subgroups of W of each index in turn until we find one which is useful. There are other methods which try to avoid computing all of these subgroups, of which there may be many. One method restricts to a special class of subgroups; another method randomly generates a fixed number of subgroups. These are given in §7.

1.2 Previous work

Over p-adic fields, there are some special cases where Galois groups can be computed.

- It is well known that the unramified extensions of K of degree d are all isomorphic, Galois and have cyclic Galois group C_d. Hence if the irreducible factors of $F(x)$ all define unramified extensions, then the splitting field of $F(x)$ is unramified, Galois and cyclic with degree $\operatorname{lcm}\{\deg g \ : \ g \in \text{Factors}(F)\}$.

- Suppose L/K is tamely ramified. Then it has a maximal unramified subfield U, and L/U is totally (tamely) ramified. It is well known that $L = U(\sqrt[e]{\zeta^r \pi})$ where $e = (L : U)$ for some uniformizer $\pi \in K$, ζ a root of unity generating U and $r \in \mathbb{Z}$. In this special form, it is straightforward to write down the splitting field and Galois group of L/K. Furthermore, it is easy to compute the compositum of tame extensions, and hence if each irreducible factor of $F(x)$ defines a tamely ramified extension, we can compute its Galois group. See §2.2.

- Greve and Pauli have studied **singly ramified** extensions, that is extensions whose ramification polygon has a single face, giving an explicit description of their splitting field and Galois group [32, Alg. 6.1]. So in particular if $F(x)$ is an Eisenstein polynomial whose ramification polygon has a single face, then we can compute its Galois group. An explicit description of this algorithm appears in Milstead's thesis [46, Alg. 3.23].

- In his thesis, Greve extends this to an algorithm for **doubly ramified** extensions [31, §6.3], that is whose ramification polygon has two faces. Essentially this uses the singly ramified algorithm for the bottom part, and class field theory and group cohomology to deal with the elementary abelian top part.

- Jones and Roberts [40] have computed all extensions of $\mathbb{Q}_p$ of degree up to 12, including their Galois group and some other invariants. These are available online in the Local Fields Database (LFDB). Some of the methods they use to compute Galois groups will feature in our general algorithm.

- Awtrey et al. have also considered degree 12 extensions of $\mathbb{Q}_2$ and $\mathbb{Q}_3$ [2]; degree 14 extensions of $\mathbb{Q}_2$ [3]; degree 15 extensions of $\mathbb{Q}_5$ [5]; and degree

16 *Galois* extensions of $\mathbb{Q}_2$ [4]. The main new idea in these articles is the **subfield Galois group content** of an extension L/K: the set of Galois groups of all proper subfields of L/K. This invariant of $\mathrm{Gal}(L/K)$ is useful in distinguishing between possible Galois groups, and is possible to compute given a database of all smaller extensions.

The difficult case appears to be when the factors of F define wildly ramified extensions whose ramification polygons have many faces.

Recently Rudzinski has developed techniques for evaluating linear resolvents [56] and Milstead has used a combination of these techniques with the ones mentioned above to compute some Galois groups in this difficult class [46].

Over global number fields $\mathcal{K}$, the forward Galois problem has a number of sophisticated solutions. These methods rely on being able to consider many different localizations, a luxury we do not have in the local case.

- A theorem of Dedekind says that if $F(x) \in \mathcal{K}[x]$ is irreducible, and so defines an extension $\mathcal{L}/\mathcal{K}$, and prime $\mathfrak{p} \lhd \mathcal{K}$ is unramified in $\mathcal{L}$, then the degrees of the irreducible factors of $F(x)$ mod $\mathfrak{p}$ give the cycle structure of an element of $\mathrm{Gal}(F)$. This is also a corollary of the Tchebotarev density theorem. This will quickly reveal if the Galois group is the full symmetric group (the "generic case" which we expect to occur with any "random" polynomial) or the alternating group. See [12].

- The version of the "Stauduhar method" [65] due to Fieker and Klüners [28] which relies on being able to choose a prime $\mathfrak{p} \lhd \mathcal{K}$ such that the splitting field of F over the completion $\mathcal{K}_\mathfrak{p}$ is a fairly low degree unramified extension. This allows for the computation in the splitting field approximately $\mathfrak{p}$-adically.

1.3 Algorithm notation

We shall be describing a highly parameterisable algorithm, and so we need some notation for its parameters. A parameter has a **name**, which is a string of characters, and possibly a sequence of **arguments**, which are themselves parameters. For example `Naive` is a parameter with name "Naive" and no

arguments, and `ARM[Global[Symmetric], All[FactorDegrees,Index]]` is a parameter with name "ARM" and two arguments.

1.4 Mathematical notation

Roman capital letters $K, L, \ldots$ denote p-adic fields. The ring of integers of K is denoted $\mathcal{O}_K$, a uniformizer is denoted π_K and the residue class field is denoted $\mathbb{F}_K = \mathcal{O}_K/(\pi_K)$. If $u \in \mathcal{O}_K$ then $\bar{u} = u + (\pi_K) \in \mathbb{F}_K$ is its residue class. We denote by v_K the valuation of $\bar{\mathbb{Q}}_p$ such that $v_K(\pi_K) = 1$.

Calligraphic capital letters $\mathcal{K}, \mathcal{L}, \ldots$ denote number fields. The ring of integers of $\mathcal{K}$ is $\mathcal{O}_\mathcal{K}$.

As introduced in §2.4, $e : W \to \mathcal{W}$ denotes a group homomorphism, and if $\mathcal{U} \leq \mathcal{W}$ is a subgroup then $q_\mathcal{U} : \mathcal{W} \to S_{\mathcal{W}/\mathcal{U}}$ denotes the action of $\mathcal{W}$ on the left cosets of $\mathcal{U}$.

As introduced in §6, s denotes a function whose input is a permutation group or a polynomial and whose output is anything. There is an equivalence relation $\sim$ on outputs such that if $F(x) \in K[x]$ then $s(\mathrm{Gal}(F)) \sim s(F)$. There may also be a partial ordering $\preceq$ on outputs such that if $H \leq G$ are groups then $s(H) \preceq s(G)$.

We may omit subscripts from the notation if they are clear from context.

1.5 A note on conjugacy

Recall that the Galois group of a polynomial $G = \mathrm{Gal}(F)$ is defined to be the group of automorphisms of the splitting field of F. Usually, we represent this as a permutation group $G \leq S_d$ where $d = \deg(F)$, such that writing the roots of F as $\alpha_1, \ldots, \alpha_d$ in some order, then G acts as $g(\alpha_i) = \alpha_{g(i)}$.

Since the order of the roots was arbitrary, G is only really defined up to conjugacy in S_d.

Sometimes, we may know more about the roots of F. For instance, if F is reducible, then G has multiple orbits. If we explicitly factorize $F = \prod_i F_i$, and let $d_i = \deg(F_i)$, then we can specify that the first d_1 roots $\alpha_1, \ldots, \alpha_{d_1}$ are the roots of F_1, the next d_2 are the roots of F_2 and so on. Letting $W = S_{d_1} \times S_{d_2} \times \ldots$ then $G \leq W \leq S_d$ is defined up to conjugacy in W. We shall see more examples in §4.

Almost everywhere in our exposition, when we talk of a group, we actually mean the conjugacy class of the group inside some understood larger group. When we talk of the collection of all groups with some property, we mean all the conjugacy classes whose groups have that property. This is to simplify the exposition.

In the implementation, a conjugacy class is usually represented by a representative group. An algorithm which returns all conjugacy classes with some property may actually return several representatives for the same class. Finding which groups generate the same class in order to remove duplicates can be computationally difficult, and so whether or not to do this, and how, is usually parameterised and usually the default is not to remove duplicates. See §11.

Henceforth, we shall typically only mention conjugacy when we have specific strategies to deal with conjugate groups.

1.6 Compendium

Most of the rest of this article describes in full detail the possible parameters to our algorithm, of which there are many. We now list the sections with the most important or novel contributions.

- §2.4: Describes the absolute resolvent method, the main focus of this article.
- §3 and §4: Methods for producing "global models" for p-adic fields, which are used to evaluate resolvents. Our constructions are more general than previous similar efforts and so can produce more efficient models.
- §5.1 and §5.3: The main two ways we perform the group theory part of deducing the Galois group. The former is to write down all possibilities and then eliminate until one remains; the latter works down the graph of possible groups using the notion of "maximal preimages of statistics" to efficiently move down the graph without blowing up the number of possibilities.
- §6.5: The main "statistic" of a resolvent we compute is the multiset of degrees of its factors. This is compared to the multiset of sizes of orbits of potential Galois groups to deduce which are possible.

- §8.3 and §9.1: Methods to produce groups from which to compute resolvents which empirically are both fast to compute and give low-degree resolvents.
- §13: The implementation, timings, performance notes, etc.

2 Galois group algorithms

This article is mainly concerned with the absolute resolvent method, introduced in §2.4. However, the algorithm is recursive, in that it may compute other Galois groups along the way, and it may suffice to use other algorithms for this purpose. Therefore, we briefly describe the other algorithms available in our implementation.

2.1 `Naive`

This explicitly computes a splitting field for $F(x)$ and explicitly computes its automorphisms.

This is the algorithm currently implemented in Magma for p-adic polynomials, called `GaloisGroup`. Since the splitting field is computed explicitly, this is only suitable when the Galois group is known in advance to be small, such as because the degree is small.

2.2 `Tame`

This requires that the factors of $F(x)$ all generate tamely ramified extensions L_i/K. The following lemma is well-known (e.g. [52, Theorem 7.2]), and allows us to put any tame extension into a very nice form.

Lemma 2.1. *If L/K is tamely ramified, with ramification degree e and residue degree f, then L may be written as $L = K(\zeta, \sqrt[e]{\zeta^r \pi})$ where ζ is a primitive (q^f-1)th root of unity, $q = |\mathbb{F}_K|$, π is a fixed uniformizer of K. $K(\zeta)$ is the maximal unramified subextension of L.*

Proof. It is well-known that $U = K(\zeta)$ is unramified of degree f, and unramified extensions are unique, so this is the maximal unramified subextension.

Then L/U is totally ramified, so a uniformizer α for L has monic minimal polynomial $E(x) = \sum_{i=0}^{e} E_i x^i \in \mathcal{O}_U[x]$ say of degree e. It is Eisenstein.

Suppose $F(x) = \sum_{i=0}^{e} F_i x^i \in \mathcal{O}_U[x]$ is also monic and Eisenstein of degree e, and that $v(F_0 - E_0) \geq 2$. Let $H(x) = F(\alpha x)\alpha^{-e} \in \mathcal{O}_L[x]$. Then

$$v(H(1)) = v(F(\alpha)) - 1 = v(F(\alpha) - E(\alpha)) - 1 = v\left(\sum_{i=0}^{e-1}(F_i - E_i)\alpha^i\right) - 1.$$

For $i = 0$, we have $v((F_0 - E_0)\alpha^0) \geq 2$ by assumption. For $i \geq 1$ we have $v((F_i - E_i)\alpha^i) \geq 1 + i/e$, and hence $v(H(1)) \geq 1/e > 0$. Also

$$v(H'(1)) = v\left(\sum_{i=1}^{e} iF_i\alpha^{i-e}\right).$$

For $i = e$, we have $v(i) = v(e) = 0$ (since $p \nmid e$ by definition of tame), $v(F_i) = 0$, $v(\alpha^{i-e}) = v(1) = 0$; and for $1 \leq i < e$ we have $v(i) \geq 0$, $v(F_i) \geq 1$ and $v(\alpha^{i-e}) = i/e - 1$. Hence the smallest term is the $i = e$ term and so $v(H'(1)) = 0$. By Hensel's lemma, we conclude that $H(x)$ and hence $F(x)$ has a root in L.

In particular, since $\bar{\zeta}$ generates $\mathbb{F}_U^\times$, there exists r such that $F(x) = x^e - \zeta^r\pi$ is of the correct form, and hence $L = U(\sqrt[e]{\zeta^r\pi})$ as claimed. □

We deduce that any tame extension of a p-adic field may be identified by the three integers (f, e, r). Algorithm 2.7 shows how to compute r. The following lemma (a slight extension of [32, Theorem 2.3]) tells us the normal closure and Galois group of such an extension.

Lemma 2.2. *Suppose $L = K(\zeta, \alpha)$ where $\alpha = \sqrt[e]{\zeta^r\pi}$ as above. Then L/K is Galois if and only if $e \mid q^f - 1$ and $e \mid r(q-1)$.*

The Galois closure has parameters $(\hat{f}, e, \hat{r})$ where $\hat{f}$ is the smallest multiple of f satisfying these conditions with $\hat{r} = r(q^{\hat{f}} - 1)/(q^f - 1)$.

If L/K is Galois, then the Galois group is generated by the automorphisms

$$\sigma : \zeta^i\alpha^j \mapsto \zeta^{qi+aj}\alpha^j$$
$$\tau : \zeta^i\alpha^j \mapsto \zeta^{i+bj}\alpha^j$$

where $a = r(q-1)/e$ and $b = (q^f - 1)/e$.

Proof. Let $U = K(\zeta)$. Since $L = U(\sqrt[e]{\zeta^r \pi})$, then L/U is Galois if and only if U contains primitive eth roots of unity, which is if and only if $e \mid q^f - 1$. $\mathrm{Gal}(U/K)$ is generated by $\zeta \mapsto \zeta^q$, and therefore L/K is Galois if and only if L/U is Galois and L also contains $\sqrt[e]{\zeta^{qr}\pi}$, which is iff L contains $\sqrt[e]{\zeta^{(q-1)r}}$, which is iff $e \mid r(q-1)$.

These arguments also show that if L/K is not Galois, then the Galois closure is got by adjoining eth roots of unity and eth roots of $\zeta^{(q-1)r}$. Since $p \nmid e$, these only increase the residue degree. If we change the residue degree f to a multiple $\hat{f}$, then the corresponding $\hat{\zeta}$ is a $(q^{f'} - 1)$th root of unity, and $\zeta = \hat{\zeta}^{(q^{\hat{f}}-1)/(q^f-1)}$, and so r becomes $\hat{r} = r(q^{\hat{f}} - 1)/(q^f - 1)$. Hence the Galois closure is got by changing f to its smallest multiple such that the result is Galois.

The group $\mathrm{Gal}(U/K)$ is generated by $\sigma : \zeta^i \mapsto \zeta^{qi}$. If L/K is Galois, we can lift σ to $\mathrm{Gal}(L/K)$ by mapping $\alpha = \sqrt[e]{\zeta^r \pi}$ to $\sqrt[e]{\zeta^{qr}\pi} = \zeta^a \alpha$. Noting that ζ^b is a primitive eth root of unity, we find that τ generates $\mathrm{Gal}(L/U)$. Since we have a generator for $\mathrm{Gal}(L/U)$ and a coset representative of a generator for $\mathrm{Gal}(L/K)/\mathrm{Gal}(L/U)$, together these generate $\mathrm{Gal}(L/K)$. □

Algorithm 2.3 (Galois closure of tame extension)**.** Given (f, e, r) defining a tame extension, and q, returns the parameters for its Galois closure.

1: **for** $\hat{f} = f, 2f, 3f, \ldots$ **do**
2: $\quad \hat{r} \leftarrow r(q^{\hat{f}} - 1)/(q^f - 1)$
3: $\quad$ **if** $e \mid q^{\hat{f}} - 1$ **and** $e \mid \hat{r}(q-1)$ **then**
4: $\quad\quad$ **return** $\hat{f}, e, \hat{r}$
5: $\quad$ **end if**
6: **end for**

If $F(x)$ is reducible, and so defines several tame extensions, we need to take their compositum in order to compute $\mathrm{Gal}(F)$.

Lemma 2.4. *If L_0/K and L/K are tame, with parameters (f_0, e_0, r_0) and (f, e, r), then L_0/K may be embedded into L/K if and only if $f_0 \mid f$, $e_0 \mid e$ and $r \equiv r_0(q^f - 1)/(q^{f_0} - 1) \mod \gcd(e_0, q^f - 1)$.*

If $L_1, \ldots, L_k/K$ are tame with parameters (f_i, e_i, r_i) then a compositum of them all is given by parameters (f, e, r) where $e = \mathrm{lcm}\{e_i\}$, f is the smallest multiple of $\mathrm{lcm}\{f_i\}$ such that there exists r such that $r \equiv r_i(q^f - 1)/(q^{f_i} - 1) \mod \gcd(e_i, q^f - 1)$ for all i.

Proof. Clearly if L_0 embeds into L then the residue and ramification degrees must divide each other. Suppose so. Then L_0 embeds into L iff $\sqrt[e_0]{\zeta_0^{r_0}\pi} \in L$. Writing $\zeta_0 = \zeta^{(q^f-1)/(q^{f_0}-1)}$ and dividing by $\sqrt[e_0]{\zeta^r\pi}$, then this occurs iff $\sqrt[e_0]{\zeta^{r_0(q^f-1)/(q^{f_0}-1)-r}}$ $\in L$. This is a root of unity, and so this holds iff there exists i such that $e_0 i \equiv r_0(q^f-1)/(q^{f_0}-1) - r \mod q^f-1$, which is to say that $r \equiv r_0(q^f-1)/(q^{f_0}-1) \mod \gcd(e_0, q^f-1)$. This proves the first claim.

For the second claim, letting L be the field defined by the parameters, then by construction L is the smallest tame field such that L_i embeds into L, and hence is a compositum. □

Algorithm 2.5 (Compositum of tame extensions)**.** Given parameters (f_i, e_i, r_i) defining tame extensions, and q, returns parameters (f, e, r) of a compositum.

1: $f_0 = \operatorname{lcm}\{f_i\}$
2: $e = \operatorname{lcm}\{e_i\}$
3: **for** $f = f_0, 2f_0, 3f_0, \ldots$ **do**
4: **if** $r \equiv r_i(q^f-1)/(q^{f_i}-1) \mod \gcd(e_i, q^f-1)$ is solvable for all i (via CRT) **then**
5: $r \leftarrow$ a solution
6: **return** f, e, r
7: **end if**
8: **end for**

Corollary 2.6. *In particular, $L_0 \cong_K L$ if and only if $f_0 = f$, $e_0 = e$ and $r_0 \equiv r \mod \gcd(e, q^f-1)$. Hence, there are $\gcd(e, q^f-1)$ isomorphism classes of extensions L/K of residue degree f and ramification degree e.*

Algorithm 2.7 (Compute r)**.** Given a tame extension L/K, returns its unique r parameter satisfying $0 \le r < g = \gcd(e, q^f-1)$.

1: $U =$ maximal unramified subextension
2: $F(x) =$ Eisenstein polynomial defining L/U
3: $\bar{\zeta} =$ a generator of $\mathbb{F}_U^\times$
4: $a = (q^f-1)/g$
5: $b = \bar{\zeta}^a$
6: $c = \overline{(F_0/\pi)}^a \in \mathbb{F}_U$

7: **return** $\log_b c$

To see that this algorithm is correct, note that we are trying to find $r \equiv r_0 \mod g$ where $F_0/\pi \equiv \bar{\zeta}^{r_0}$. Now $r \equiv r_0 \mod g$ if and only if $q^f - 1 \mid (r - r_0)a$ if and only if $\bar{\zeta}^{(r-r_0)a} = 1$ if and only if $b^r \equiv c$. Also note that the final discrete logarithm is in a group of order g, so is quick to compute (e.g. by exhaustion) even when q is large.

We now present the algorithm to compute the Galois group of a polynomial $F(x)$ whose factors define tame extensions. First we factorize F, compute the extensions L_i/K corresponding to its factors, and find the parameters (f_i, e_i, r_i). Next, we use the above lemmas to find parameters for the compositum of the Galois closure of these fields, and write down generators for this group as functions on (i, j) representing $\zeta^i \alpha^j$. We use these to produce generators for the group as permutations on roots of F, observing that $\zeta_i = \zeta^{c_i}$ and $\alpha_i = \zeta^{c'_i} \alpha^{e/e_i}$ for c_i and c'_i as given in the algorithm.

Algorithm 2.8 (Galois group: tame)**.** Given a polynomial $F(x)$ whose factors define tame extensions, returns its Galois group.

1: $(F_1, \ldots, F_k) \leftarrow$ factorization of F
2: $(f_i, e_i, r_i) \leftarrow$ parameters of extensions defined by each F_i (Alg. 2.7)
3: $(f, e, r) \leftarrow$ parameters of Galois closure of compositum (Alg. 2.5 and 2.3)
4: $a \leftarrow r(q-1)/e \in \mathbb{Z}$
5: $b \leftarrow (q^f - 1)/e \in \mathbb{Z}$
6: $s \leftarrow$ the function $(i, j) \mapsto (qi + aj \bmod q^f - 1, j)$
7: $t \leftarrow$ the function $(i, j) \mapsto (i + bj \bmod q^f - 1, j)$
8: **for** $i = 1, \ldots, k$ **do**
9: $\quad c_i \leftarrow (q^f - 1)/(q^{f_i} - 1)$
10: $\quad c'_i \leftarrow (c_i r_i - r)/e_i$
11: $\quad x \leftarrow ((c_i, 0), (c'_i, e/e_i))$
12: $\quad X \leftarrow$ the orbit of x under iterating s and t (size $d_i = f_i e_i = \deg F_i$)
13: $\quad \sigma_i \leftarrow$ the permutation of X induced by s
14: $\quad \tau_i \leftarrow$ the permutation of X induced by t
15: **end for**
16: $\sigma \leftarrow (\sigma_1, \ldots, \sigma_k) \in S_{d_1} \times \ldots \times S_{d_k}$

17: $\tau \leftarrow (\tau_1, \ldots, \tau_k) \in S_{d_1} \times \ldots \times S_{d_k}$
18: **return** $\langle \sigma, \tau \rangle$

2.3 `SinglyRamified`

This computes the Galois group of $F(x)$ provided it is irreducible and defines an extension whose ramification filtration contains a single segment. Such an extension is called **singly ramified**.

When the extension is tamely ramified, we can use the `Tame` algorithm. Otherwise the extension is totally wildly ramified and we use an algorithm due to Greve and Pauli [32, Alg. 6.1]. An explicit description is given by Milstead [46, Alg. 3.23].

2.4 `ARM`: Absolute Resolvent Method

The absolute resolvent method is the focus of the remainder of this article and is based on the following simple lemma.

Lemma 2.9. *Suppose* $G := \mathrm{Gal}(F) \leq W \leq S_d$ *where* $d = \deg F$*, and take any* $U \leq W$*. Now* S_d *acts on* $\mathbb{Z}[x_1, \ldots, x_d]$ *by permuting the variables, so suppose* $I \in \mathbb{Z}[x_1, \ldots, x_n]$ *such that* $\mathrm{Stab}_W(I) = U$ *(we say* I *is a* ***primitive*** W***-relative*** U***-invariant****). Letting* $\alpha_1, \ldots, \alpha_d$ *be the roots of* F*, define* $\beta_{wU} = wU(I)(\alpha_1, \ldots, \alpha_n)$ *(this is well-defined since* I *is fixed by* U*) and define the* ***resolvent*** $R(t) := \prod_{wU \in W/U}(t - \beta_{wU})$*. Then* $R(t) \in K[t]$*. If* R *is squarefree, then its Galois group corresponds to the coset action of* G *on* U*. That is, letting* $q : W \to S_{W/U}$ *be the coset action, then identifying* $wU \leftrightarrow \beta_{wU}$ *we have* $\mathrm{Gal}(R) = q(G)$*.*

Proof. Writing $R(t) := \tilde{R}(\alpha_1, \ldots, \alpha_d; t)$ where

$$\tilde{R}(x_1, \ldots, x_d; t) := \prod_{wU \in W/U} (t - wU(I)(x_1, \ldots, x_d))$$

then the t-coefficients of $\tilde{R}$ are fixed by W (the action of W re-orders the product) and hence by G. We conclude that the t-coefficients of R are fixed by G too, and hence by Galois theory $R(t) \in K[t]$.

If R is squarefree, then there is a 1-1 correspondence between the cosets $\{wU\}$ of W/U and the roots $\{\beta_{wU}\}$ of R. Take $g \in G$, then

$$\begin{aligned} g(\beta_{wU}) &= g(wU(I)(\alpha_1, \ldots, \alpha_d)) \\ &= wU(I)(g(\alpha_1), \ldots, g(\alpha_d))) \\ &= wU(I)(\alpha_{g(1)}, \ldots, \alpha_{g(d)}) \\ &= gwU(I)(\alpha_1, \ldots, \alpha_d) \\ &= \beta_{gwU} \end{aligned}$$

so the action of G on the roots of R corresponds to the coset action, as claimed. □

Therefore, if we have some W containing G and a means to compute resolvents R for $U \leq W$, then since $\mathrm{Gal}(R) = q(G)$ is a function of G, we can deduce information about G by finding some information about $\mathrm{Gal}(R)$. Specifically how we compute resolvents and deduce information about G is controlled by two parameters.

Parameters.

- A resolvent evaluation algorithm (§3) selects a fixed group $W \leq S_d$ such that $G \leq W$, and thereafter is responsible for evaluating the resolvents $R(t)$ from selected $U \leq W$ and invariants $I \in \mathbb{Z}[x_1, \ldots, x_d]$.
- A group theory algorithm (§5) is responsible for deducing the Galois group G by choosing a suitable U, and then using the resolvent R returned by the resolvent evaluation algorithm to gather information about G.

In fact, we generalize the situation a little. The resolvent evaluation algorithm actually selects a group homomorphism $e : W \to \mathcal{W}$ such that: $G \leq W \leq S_d$; $\mathcal{W} \leq S_{d'}$; given $\mathcal{U} \leq \mathcal{W}$, $\mathrm{Gal}(R)$ is the coset action of $e(G)$ on $\mathcal{U}$; and we can evaluate resolvents relative to $\mathcal{W}$. We call e an **overgroup embedding**. This generalization allows for more freedom in global models, as explained in §4.

Algorithm 2.10 (Galois group: absolute resolvent method)**.** Given a polynomial $F(x) \in K[x]$, returns its Galois group.

1: Initialize the resolvent evaluation algorithm.
2: Let $e : W \to \mathcal{W}$ be the overgroup embedding selected.
3: Initialize the group theory algorithm.
4: If we have determined the Galois group, then return it.
5: Let $\mathcal{U}$ be a subgroup of $\mathcal{W}$.
6: Let I be a primitive $\mathcal{W}$-relative $\mathcal{U}$-invariant.
7: Let R be the resolvent corresponding to I.
8: Use R to deduce information about the Galois group.
9: Go to step 4.

The resolvent algorithm controls steps 1, 2 and 7. The group theory algorithm controls steps 3, 4, 5 and 8. Step 6 could also be parameterised, but we find it is sufficient to use the algorithm due to Fieker and Klüners [28, §5], implemented as the intrinsic `RelativeInvariant` in Magma.

Remark 2.11. Using resolvents to compute Galois groups is not new. Stauduhar's method [65] for polynomials over $\mathbb{Q}$ computes resolvents relative to S_d by computing complex approximations to the roots. This was improved by Fieker and Klüners [28] to a "relative resolvent method" which allows the overgroup W to be made smaller at each iteration until it equals G. Over $\mathbb{Q}_p$, an absolute resolvent method has been used by Jones and Roberts [40] to compute the Galois group of fields of degree up to 12, computing resolvents in $W = S_{d_2} \wr S_{d_1}$ corresponding to a subfield of degree d_1.

2.5 `Sequence`

This algorithm takes as parameters a sequence of other algorithms to compute Galois groups. It tries each algorithm in turn until one succeeds. This is mainly useful to deal with special cases first (e.g. `Tame` or `SinglyRamified`) before applying a general method (e.g. `ARM`).

3 Resolvent evaluation algorithms

These are used as part of the `ARM` (absolute resolvent method) algorithm for computing Galois groups. They are responsible for selecting an overgroup

embedding $e : W \to \mathcal{W}$ such that $G \leq W$ and thereafter evaluating resolvents relative to $\mathcal{W}$. If R is a resolvent with respect to $\mathcal{U} \leq \mathcal{W}$, and $q : \mathcal{W} \to S_{\mathcal{W}/\mathcal{U}}$ is the corresponding coset action, then e must satisfy $\mathrm{Gal}(R) = q(e(G))$.

Currently there is one option, `Global`, described below.

3.1 `Global`

Definition 3.1. A **global model** for a p-adic field K is an embedding $i : \mathcal{K} \to K$ where $\mathcal{K}$ is a global number field such that K is a completion of $\mathcal{K}$ and i is the corresponding embedding.

If L/K is an extension of p-adic fields, and $i : \mathcal{K} \to K$ is a global model for K, then a **global model for L/K extending** i is a global model $j : \mathcal{L} \to L$ of L such that $j|_{\mathcal{K}} = i$.

Similarly a **global model for $F(x) \in K[x]$ extending** i is $\prod_k \mathcal{F}_k$ where $F = \prod_k F_k$ is the factorization over K of F into irreducible factors, L_k/K are the corresponding extensions, $i_k : \mathcal{L}_k \to L_k$ are global models for L_k/K extending i, and $\mathcal{L}_k \cong \mathcal{K}(x)/(\mathcal{F}_k(x))$.

We shall often refer to $\mathcal{K}$ itself as the global model, instead of the embedding i.

The `Global` algorithm computes a global model $\mathcal{K}$ for K and a global model $\mathcal{F}(x) \in \mathcal{K}[x]$ for the input $F(x) \in K[x]$ extending $\mathcal{K}$. At the same time, it computes the required overgroup embedding $e : W \to \mathcal{W}$ such that not only $G \leq W$ but also $\mathrm{Gal}(\mathcal{F}/\mathcal{K}) \leq \mathcal{W}$ and $e(G)$ is the decomposition group.

For irreducible F defining L/K of degree d and $\mathcal{F}$ defining $\mathcal{L}/\mathcal{K}$ of degree $d' = sd$ then we will typically have a tower $\mathcal{L}/\mathcal{K}'/\mathcal{K}$ such that $(\mathcal{L} : \mathcal{K}') = d$, $(\mathcal{K}' : \mathcal{K}) = s$ and $\mathcal{K}'$ has s completions $i_1, \ldots, i_s : \mathcal{K}' \to K$ extending i, extending uniquely to s completions $j_1, \ldots, j_s : \mathcal{L} \to L$. See Figure 1. Then $\mathrm{Gal}(\mathcal{L}/\mathcal{K}) \leq S_d \wr S_s$, with the decomposition groups of the s completions $j_1, \ldots, j_s$ being the embedding of $\mathrm{Gal}(L/K)$ into the s copies of S_d in $S_d \wr S_s$. Hence $\mathrm{Gal}(L/K)$ acts on $\mathcal{L}$ as the diagonal embedding of $\mathrm{Gal}(L/K)$ into $S_d^s \leq S_d \wr S_s$, and $e : W \to \mathcal{W}$ will be such a diagonal embedding.

Definition 3.2. In this scenario, we say $j_1, \ldots, j_s : \mathcal{L} \to L$ is a **global model for L/K extending i of index s.**

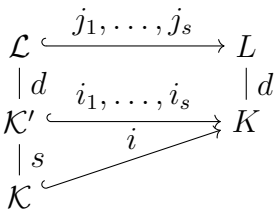

Figure 1: A typical global model extension.

Remark 3.3. We allow $s > 1$ to give our global model constructions more freedom. The `RootOfUnity` and `SinglyWild` constructions (§4) would not be possible in general if we restricted to $s = 1$.

The algorithm then can evaluate resolvents as follows. For each complex embedding $c : \mathcal{K} \to \mathbb{C}$, we compute the roots of $c(\mathcal{F})$ to high precision. Letting $\tilde{\alpha}_1, \ldots, \tilde{\alpha}_{d'}$ be these roots, we compute

$$\tilde{R}_c(t) := \prod_{w\mathcal{U} \in \mathcal{W}/\mathcal{U}} (t - w\mathcal{U}(I)(\tilde{\alpha}_1, \ldots, \tilde{\alpha}_{d'}))$$

which is an approximation to $c(R(t)) \in \mathbb{C}[t]$.

We can always arrange for $\mathcal{F}(x)$ to be monic and integral, so that its roots are integral, and therefore $R(t) \in \mathcal{O}_{\mathcal{K}}[t]$. Firstly, suppose that $\mathcal{K} = \mathbb{Q}$ (so $K = \mathbb{Q}_p$), then we know $R(t) \in \mathbb{Z}[t]$ and therefore assuming we have computed $\tilde{R}(t)$ sufficiently precisely, then we can compute $R(t)$ by rounding its coefficients to the nearest integer.

More generally, for each coefficient R_i of $R(t)$ we take the vector $(\tilde{R}_{c,i})_c$ which should be a close approximation to $(c(R_i))_c$. Since R_i are integral, $(c(R_i))_c$ is an element of the **Minkowski lattice** $\prod_c c(\mathcal{O}_{\mathcal{K}})$, which is discrete, and therefore we can deduce R_i by rounding $(\tilde{R}_{c,i})_c$ to the nearest point in the lattice. This can be done using lattice basis reduction techniques such as LLL.

Parameters.

- A global model algorithm (§4) which specifies how to produce a global model for $F(x)$.

Algorithm 3.4 (Resolvent: `Global`)**.** Given a global model $\mathcal{F}(x) \in \mathcal{K}[x]$ and

subgroup $\mathcal{U} \leq \mathcal{W}$, returns the corresponding resolvent $R(t)$.

1: Choose a Tschirnhaus transformation $T \in \mathbb{Z}[x]$ (see Rmk. 3.5).
2: Choose a complex floating point precision, k decimal digits (see Rmk. 3.6).
3: Compute complex approximations to the roots of $c(\mathcal{F})$ for each complex embedding $c : \mathcal{K} \to \mathbb{C}$.
4: Compute $\tilde{R}_c(t) = \prod_{w\mathcal{U} \in \mathcal{W}/\mathcal{U}}(t - w\mathcal{U}(I)(T(\tilde{\alpha}_1), \ldots, T(\tilde{\alpha}_{d'})))$.
5: Round $(\tilde{R}_{c,i})_i$ to the nearest point of the Minkowski lattice of $\mathcal{O}_\mathcal{K}$, and let R_i be the corresponding element of $\mathcal{O}_\mathcal{K}$.
6: If $R(t) \in \mathcal{K}[t]$ is not squarefree, go to Step 1.
7: Return $R(t)$.

Remark 3.5. In Step 1, a Tschirnhaus transformation is any randomly selected polynomial in $\mathbb{Z}[x]$. Its purpose is to ensure that $R(t)$ is squarefree. Indeed, if $R(t)$ is not squarefree, then there is some coincidence between its roots, and therefore some unintended structure between the roots of F. By transforming the roots, we should destroy this structure.

Such a transformation always exists [30]. In practice, it suffices to use $T(x) = x$ initially, and thereafter to choose a random polynomial of small degree and coefficients, increasing the degree and coefficient bound at each iteration.

Remark 3.6. It is important in Step 2 that we choose a complex floating point precision k such that the rounding step produces the correct answer. We do this as follows.

First, we find an upper bound on the absolute valuations of the roots of $c(\mathcal{F})$ for each complex embedding c. In principle this could be done by analyzing the polynomials which define the global model and bounding their roots in terms of the coefficients, but in our current implementation we instead compute the complex roots to some default precision (30 decimal digits) and take the size of the largest root as our bound. It is possible although unlikely that the latter approach introduces enough precision error that this bound is incorrect, and hence this part of the implementation does not yield proven results.

Using this upper bound, we can follow through the computation of $\tilde{R}_c$ to get upper bounds on its coefficients. By increasing the bounds by a small fraction at each computation, we can absorb the effect of any complex precision error. We

then select a precision so that the absolute errors on the coefficients $\tilde{R}_{c,i}$ are less than half the shortest distance between two elements of the Minkowski lattice. We then add a generous margin to the precision (say 20 decimal digits) so that we can check in the code that we are in fact very close (say within 10 decimal digits) of an integer point.

Remark 3.7. The choice to approximate the roots of $\mathcal{F}$ in the complex field $\mathbb{C}$ is somewhat arbitrary. We could instead pick a prime ℓ such that $\mathcal{F}$ has a small splitting field over $\mathbb{Q}_\ell$ and approximate the roots ℓ-adically. Making such a change usually improves the reliablility and precision requirements. The theory of the Minkowski lattice carries over into this setting.

4 Global model algorithms

Given a polynomial $F(x) \in K[x]$ and a global model $i_1, \ldots, i_r : \mathcal{K} \to K$, a global model algorithm computes a global model $\mathcal{F}(x)$ for $F(x)$ extending $\mathcal{K}$. It also computes an overgroup embedding $e : W \to \mathcal{W}$ such that $G \leq W$ and $\mathrm{Gal}(\mathcal{F}/\mathcal{K}) \leq \mathcal{W}$ and $e(G) \leq \mathrm{Gal}(\mathcal{F}/\mathcal{K})$ is the action of G on the roots of $\mathcal{F}$.

Note that for ease of notation we typically describe the algorithms in the case of a single completion, i.e. $r = 1$. It is usually trivial to extend these to $r > 1$ by repeating the same procedure for each completion individually and combine them with the Chinese remainder theorem to ensure the right properties hold with respect to all completions simultaneously.

4.1 `Symmetric`

Given irreducible $F(x) \in K[x]$, this finds a polynomial $\mathcal{F}(x) \in \mathcal{K}[x]$ sufficiently close to $F(x)$ (along all embeddings $i_1, \ldots, i_r$) that they have the same splitting field over K. Generically we expect that $\mathrm{Gal}(\mathcal{F}/\mathcal{K}) = S_d$, since we are not imposing any further restriction of $\mathcal{F}$, and therefore the corresponding overgroup embedding is taken to be the identity $e : S_d \to S_d$.

To find such a polynomial, we pick some precision parameter $k \in \mathbb{N}$. We take for each i_j some polynomial $\mathcal{F}_j(x) \in \mathcal{K}[x]$ which "approximates $F(x)$ along i_j" meaning that $i_j(\mathcal{F}_j(x)) - F(x)$ has coefficients of valuation at least k. Next we use

the Chinese remainder theorem on $\mathcal{F}_j$ mod $\mathfrak{p}_j^k$ to find $\mathcal{F}$ such that $i_j(\mathcal{F}(x)) - F(x)$ has coefficients of valuation at least k, and then we check that $\mathcal{F}$ is a global model. If not, we increase k. By keeping k small, we limit the size of the coefficients of $\mathcal{F}$, which in turn limits the precision required in the complex arithmetic later.

Algorithm 4.1 (Global model: `Symmetric`)**.** Given irreducible $F(x) \in K[x]$, global model $i_1, \ldots, i_r : \mathcal{K} \to K$ corresponding to $\mathfrak{p}_1, \ldots, \mathfrak{p}_r$, computes $\mathcal{F}(x) \in \mathcal{K}[x]$ such that for all $j = 1, \ldots, r$, $i_j(\mathcal{F}(x))$ defines a field isomorphic to that generated by F.

1: $L \leftarrow$ the field generated by F.
2: **for all** $k = 1, 2, 4, 8, \ldots$ **do**
3: **for all** $j = 1, \ldots, r$ **do**
4: $\mathcal{F}_j \leftarrow$ an approximation to $F(x)$ along i_r to precision k
5: **end for**
6: $\mathcal{F} \leftarrow \mathrm{CRT}\left\{(\mathcal{F}_j, \mathfrak{p}_j^k) : j = 1, \ldots, r\right\}$
7: **if** $i_j(\mathcal{F}(x))$ are all irreducible and have roots in L **then**
8: **return** $\mathcal{F}(x)$
9: **end if**
10: **end for**

Parameters.

- An optional Galois group algorithm (§2), which is used to compute the Galois group of F. This does not help with computing resolvents, but the information is stored in our representation of the embedding $e : W \to \mathcal{W}$ so that the group theory algorithms can cut down their work. This can be useful when `Symmetric` appears as part of a larger global model algorithm; e.g. in `RamTower[Symmetric]` (§4.3), the `Symmetric` global model algorithm runs only on polynomials defining a singly ramified extension, and therefore `RamTower[Symmetric[SinglyRamified]]` will compute the Galois group of each piece in the tower.

4.2 `Factors`

This factorizes $F(x) = \prod_k F_k(x)$ into irreducible factors over K, produces a global model $\mathcal{F}_k(x)$ for each factor, and then the global model is $\mathcal{F}(x) = \prod_k \mathcal{F}_k(x)$.

If $e_k : W_k \to \mathcal{W}_k$ are the corresponding overgroup embeddings for the factors, then the overgroup embedding is the direct product $\prod_k e_k : \prod_k W_k \to \prod_k \mathcal{W}_k$.

Parameters.

- A global model algorithm (§4), which is used to compute the global model for each factor.

4.3 `RamTower`

Assuming $F(x)$ is irreducible and defines an extension L/K, this finds the ramification filtration $L = L_t/\ldots/L_0 = K$ of L/K. For each segment L_k/L_{k-1}, it produces a global model extending the global model of the segment below it. Then the global model is the final model in this iteration.

If $e_k : W_k \to \mathcal{W}_k$ are the overgroup embeddings corresponding to the model for each segment, then the overgroup embedding for F is the wreath product $(e_t \wr \cdots \wr e_1) : (W_t \wr \cdots \wr W_1) \to (\mathcal{W}_t \wr \cdots \wr \mathcal{W}_1)$. In the case where the global model for L_i/L_{i-1} is of index $s > 1$, then the corresponding wreath product will also include a diagonal embedding.

Parameters.

- A global model algorithm (§4), which is used to compute the global model for each segment.

4.4 `D4Tower`

Assuming $F(x)$ is irreducible and defines an extension L/K with Galois group $D_4 = C_2 \wr C_2$, this finds the unique quadratic subfield $K \subset M \subset L$, finds a symmetric global model for M/K and then a symmetric global model for L/M extending it. The overgroup embedding is the identity $e : C_2 \wr C_2 \to C_2 \wr C_2$.

See §13.6 for an application.

4.5 `RootOfUnity`

Assuming the splitting field L of F over K is unramified, and therefore generated by a primitive nth root of unity ζ, we define the global model to be $\mathcal{L} = \mathcal{K}(\zeta)$.

We naturally identify $\mathcal{W} = \mathrm{Gal}(\mathcal{L}/\mathcal{K})$ with a subgroup of $(\mathbb{Z}/n\mathbb{Z})^\times$, identifying $i \bmod n$ with $\zeta \mapsto \zeta^i$. The subgroup $W = \langle q \rangle \leq \mathcal{W}$ is the decomposition group, i.e. $\mathrm{Gal}(L/K)$, and we define $\mathcal{K}'$ to be its fixed field. Let $i_1, \ldots, i_s$ be all the embeddings $\mathcal{K}' \to K$, where $s = (\mathcal{K}' : \mathcal{K}) = (\mathcal{W} : W)$. For each of these, we get unique embeddings $j_1, \ldots, j_s : \mathcal{L} \to L$ giving our model of index s.

Now $\mathcal{W}$ naturally faithfully acts on $|\mathcal{W}| = (\mathcal{L} : \mathcal{K}) = sd$ elements of $(\mathbb{Z}/n\mathbb{Z})^\times$, where $i \bmod n$ is now identified with ζ^i. Separating these elements out into orbits under W (i.e. the cosets of W, of size d), we find that we can identify $\mathcal{W}$ as a subgroup of $S_d \wr S_s$. Under this identification, the s embeddings $j_1, \ldots, j_s : \mathcal{L} \to L$ correspond to the s copies of S_d in the wreath product, and so the corresponding overgroup embedding $e : W \to \mathcal{W}$ is the restriction of the diagonal embedding $S_d \to S_d^s \to S_d^s \rtimes S_s = S_d \wr S_s$.

Parameters.

- `Minimize` which is true or false. When false, we use $n = q^d - 1$. When true, n is the smallest divisor of $q^d - 1$ not dividing $q^c - 1$ for any $c < d$.
- `Complement` which is true or false. When true, we search for a **complement** to W — i.e. a subgroup $H \leq \mathcal{W}$ such that $H \cap W = 1$ — of smallest index possible, and then replace $\mathcal{L}$ by the fixed field of H. By design, this still has a completion to L, but now we have $sd = (W : H)$ and therefore s is as small as possible.

Remark 4.2. The complement option usually finds a **perfect complement**, i.e. such that $\langle H, W \rangle = \mathcal{W}$, and hence $s = 1$ (which is optimal). For example, suppose $\mathcal{K} = \mathbb{Q}$ and $K = \mathbb{Q}_p$, $p \leq 7$ and $d \leq 50$, then there is a perfect complement unless: $p = 2$ and $8 \mid d$; or $p = 3$ and $d = 9$; or $p = 7$ and $d \in \{5, 8\}$.

Remark 4.3. The Grunwald–Wang theorem of class field theory [1, Ch. X, §2] implies that if K is a completion $\mathcal{K}_\mathfrak{p}$, and L/K is cyclic, degree d, then there is $\mathcal{L}/\mathcal{K}$ cyclic of degree d which completes to L. There is an exception at primes $\mathfrak{p} \mid 2$ and degrees $8 \mid d$, for which $(\mathcal{L} : \mathcal{K}) = 2d$ is sometimes necessary.

4.6 `RootOfUniformizer`

Assuming F is irreducible of degree d over K and defines a totally tamely ramified extension L/K, then by Lemma 2.1 we have $L = K(\sqrt[d]{\pi})$ for some uniformizer

$\pi \in K$. Taking a sufficiently precise approximation to π (along all embeddings $\mathcal{K} \to K$), we may assume that $\pi \in \mathcal{K}$, and we define the global model to be $\mathcal{L} = \mathcal{K}(\sqrt[d]{\pi})$. Each embedding of $\mathcal{K} \to K$ extends uniquely to $\mathcal{L} \to L$.

Letting ζ be a primitive dth root of unity, then clearly $\mathcal{K}(\sqrt[d]{\pi}, \zeta)$ is the normal closure and its Galois group $\mathcal{W}$ (which is a function of $\mathrm{Gal}(\mathcal{K}(\zeta)/\mathcal{K})$ which may be computed explicitly) acts faithfully on the d elements $\sqrt[d]{\pi}$, $\zeta\sqrt[d]{\pi}$, …, $\zeta^{d-1}\sqrt[d]{\pi}$. We take $W = \mathcal{W}$ and the overgroup embedding to be the identity map.

4.7 `SinglyWild`

Suppose $F(x) \in K[x]$ defines a singly wildly ramified extension L/K of degree $d = p^k$. That is, a totally wildly ramified extension whose ramification polygon has a single face. By the results of Greve [32, Theorem 7.3], there is a Galois tame extension T/K such that LT/K is the Galois closure of L/K. In particular LT/T is Galois with group C_p^k. Moreover, T depends only on the ramification polygon and its residual polynomials. We now describe a construction for a global model for L/K.

Let $\mathcal{T}/\mathcal{K}$ be a global model for T/K which is Galois and let $\mathcal{K}'$ be the fixed field of the decomposition group, so $\mathrm{Gal}(\mathcal{T}/\mathcal{K}') = \mathrm{Gal}(T/K)$.

Assume (see Remark 4.4) that $\zeta_p \in T$, then LT/T is a Kummer extension and so $LT = T(\sqrt[p]{a_1}, \ldots, \sqrt[p]{a_k})$ for some $a_i \in T$. The fact that LT/K is Galois is equivalent to saying the group $\langle a_1, \ldots, a_k \rangle \leq T^\times/(T^\times)^p$ is stabilized by $\mathrm{Gal}(T/K)$.

By taking sufficiently precise approximations, we may assume $a_i \in \mathcal{T}$. Assume $\zeta_p \in \mathcal{T}$ also. Now there is no reason to expect that $\langle a_1, \ldots, a_k \rangle \leq \mathcal{T}^\times/(\mathcal{T}^\times)^p$ is stabilized by $\mathrm{Gal}(\mathcal{T}/\mathcal{K}')$, and hence $\mathcal{T}(\sqrt[p]{a_1}, \ldots, \sqrt[p]{a_k})/K$ is probably not Galois.

Define

$$A = \left\{ a_1^{i_1} \cdots a_k^{i_k} \,:\, 0 \leq i_1, \ldots, i_k < p \right\} \subset \mathcal{T}$$

and observe that A is a set of representatives for $\langle a_1, \ldots, a_k \rangle \leq T^\times/(T^\times)^p$. For $g \in \mathrm{Gal}(\mathcal{T}/\mathcal{K}')$ and $a \in A$ define $g \cdot a$ to be the unique $b \in A$ such that $g(a) \equiv b \bmod (T^\times)^p$.

For $a \in A$ define

$$a' = \prod_{g \in \mathrm{Gal}(\mathcal{T}/\mathcal{K}')} g^{-1}(g \cdot a) \in \mathcal{T}$$

and

$$A' = \{a' \,:\, a \in A\} \subset \mathcal{T}.$$

By definition $(g\cdot a) \equiv g(a) \mod (T^\times)^p$ and so $g^{-1}(g\cdot a) \equiv a$ and so $a' \equiv a^{(T:K)}$. If we further assume (see Remark 4.4) that $p \nmid (T : K)$ then A' is also a set of representatives.

Observe that if $g, h \in \mathrm{Gal}(\mathcal{T}/\mathcal{K}')$ then by definition $gh(a) \equiv (gh \cdot a)$ and $g(h(a)) \equiv (g \cdot (h \cdot a))$ and so $gh \cdot a = g \cdot (h \cdot a)$. We deduce

$$\begin{aligned} h(a') &= h(\prod_g g^{-1}(g \cdot a)) \\ &= \prod_g h((gh)^{-1}(gh \cdot a)) \qquad \text{changing variables } g \mapsto gh \\ &= \prod_g g^{-1}(g \cdot (h \cdot a)) \\ &= (h \cdot a)'. \end{aligned}$$

So A' is stabilized by $\mathrm{Gal}(\mathcal{T}/\mathcal{K}')$.

Also observe that if $a, b \in A$ then by definition $g(ab) \equiv (g \cdot ab)$ and $g(ab) = g(a)g(b) \equiv (g \cdot a)(g \cdot b)$ and so $g \cdot ab \equiv (g \cdot a)(g \cdot b) \mod (T^\times)^p$. Now since all these terms are in A then in fact

$$g \cdot ab \equiv (g \cdot a)(g \cdot b) \mod \langle a_1^p, \ldots, a_k^p \rangle \subseteq (\mathcal{T}^\times)^p.$$

We deduce $(ab)' \equiv a'b' \mod (\mathcal{T}^\times)^p$ and so

$$\langle A' \rangle (\mathcal{T}^\times)^p = \langle a_1', \ldots, a_k' \rangle (\mathcal{T}^\times)^p.$$

Therefore letting

$$\mathcal{N} = \mathcal{T}\left(\sqrt[p]{a_1'}, \ldots, \sqrt[p]{a_k'}\right)$$

then $\mathcal{N}/\mathcal{T}$ is the Kummer extension corresponding to $\langle A' \rangle (\mathcal{T}^\times)^p/(\mathcal{T}^\times)^p$ and $\mathcal{N}/\mathcal{K}'$ is Galois.

By construction we have $\mathrm{Gal}(\mathcal{N}/\mathcal{K}') = \mathrm{Gal}(LT/K)$ and so by letting $\mathcal{L}$ be the subfield of $\mathcal{N}$ fixed by $\mathrm{Gal}(LT/L)$ then $\mathcal{L}/\mathcal{K}$ is a global model for L/K.

Each embedding $i : \mathcal{K} \to K$ extends to $s = (\mathcal{K}' : \mathcal{K})$ embeddings $i_1, \ldots, i_s$:

$\mathcal{K}' \to K$, extending uniquely to $j_1, \ldots, j_s : \mathcal{L} \to L$.

We take $W = \mathrm{Gal}(LT/K)$, $\mathcal{W} = W \wr \mathrm{Gal}(\mathcal{K}'/\mathcal{K})$ and $e : W \to \mathcal{W}$ the diagonal embedding.

Remark 4.4. This construction relies on two assumptions which do not hold in general: $\zeta_p \in \mathcal{T}$ and $p \nmid (\mathcal{T} : \mathcal{K}') = (T : K)$.

Define K_0 to be the extension of K containing ζ_p and whose residue degree is a sufficiently large power of p. Then $\mathcal{K}(\zeta_p)$ is a global model for $K(\zeta_p)$ and then taking any global model for $K_0/K(\zeta_p)$ (which is unramified) we get a global model for K_0/K.

The assumptions now hold for LK_0/K_0, and hence we can produce a global model for LK_0/K.

Now considering any extension L/K, we can produce K_0/K in this manner such that the assumptions hold for any singly ramified piece of L/K, and therefore we can produce a global model for LK_0/K, and hence compute $\mathrm{Gal}(LK_0/K)$. Letting H be the subgroup $\mathrm{Gal}(LK_0/L)$, then $\mathrm{Gal}(L/K)$ is the image of the coset action on H.

Remark 4.5. We could take $\mathcal{N}$ itself to be the global model instead of $\mathcal{L}$, since L/K and LT/K have the same Galois closure.

Remark 4.6. Due to time constraints, we have not implemented `SinglyWild` in this generality. We have only implemented the special case $p = 2$ and $(T : K) = 1$, for which the assumptions trivially hold. See §13.9.

4.8 `Select`

This selects between several different global model algorithms. It takes as parameters pairs of expressions and global model algorithms, and a final "default" global model algorithm. An expression may have some free variables, and must evaluate to either true or false. The first such expression to evaluate to true, we use the corresponding global model algorithm. If none are true, we use the default.

The expressions may have the polynomial F as a free variable. They may also have some derived information about F, such as `unram` which is true iff F defines an unramified extension, `tame` which is true iff F defines a tame extension, and so on.

For example, the algorithm

```
Select[unram,RootOfUnity][tame,RootOfUniformizer][SinglyWild]
```

is equivalent to `RootOfUnity`, `RootOfUniformizer` or `SinglyWild` depending on whether F defines an unramified, tame, or wild extension.

5 Group theory algorithms

The job of a group theory algorithm is to decide, given the overgroup embedding $e : W \to \mathcal{W}$, which subgroups $\mathcal{U} \leq \mathcal{W}$ to form resolvents from, and to use those resolvents to deduce the Galois group $G \leq W$.

We recommend now reading the definition of statistic at the start of §6. A statistic is our means of comparing groups with resolvents.

5.1 `All`

This algorithm proceeds by writing down all possible Galois groups G (up to W-conjugacy), and then eliminating possibilities until only one remains.

Parameters.

- A statistic algorithm (§6) which determines which properties of the Galois groups G and resolvents R to compare.
- A subgroup choice algorithm (§7) which determines how we choose a subgroup $\mathcal{U}$.

The subgroup choice algorithm is used to choose a subgroup $\mathcal{U}$. Then, given a resolvent R, we use the statistic algorithm to compute a statistic $s(\mathrm{Gal}(R))$ and see for which G in the list of possible Galois groups this equals $s(q(e(G)))$ where q is the coset action of $\mathcal{W}$ on $\mathcal{W}/\mathcal{U}$. We eliminate the G for which the statistics differ. We are done when only one G remains.

Remark 5.1. The parameters must be chosen correctly to ensure that the algorithm terminates, otherwise it is possible that the subgroup choice algorithm cannot find a useful subgroup for the given statistic. Lemma 5.4 implies the algorithm terminates for the `HasRoot` statistic (or any more precise statistic such

as `FactorDegrees`) and the `Tranche:All` or `Tranche:Index` subgroup choice algorithm.

5.2 `Maximal`

This algorithm avoids the need to enumerate all possible Galois groups. We start at the top of the directed acyclic graph of subgroups of W and work our way down, at each stage either proving that a current group under consideration is not the Galois group, and so moving on to its maximal subgroups, or proving that the Galois group is not a subgroup of some of the maximal subgroups of a group under consideration.

Parameters.

- A statistic algorithm (§6).
- A subgroup choice algorithm (§7).
- `Descend` (see later in this section).
- `Usefulness` (see later in this section).

Specifically, at all times we have a set $\mathcal{P}$ of subgroups of W such that we know that the Galois group is contained in at least one of them. We call this the **pool**. Initially we have $\mathcal{P} = \{W\}$. If for some resolvent R and $P \in \mathcal{P}$ we find that their statistics do not agree, i.e. $s(R) \not\sim s(q(e(P)))$, then we record that $G \neq P$. We also test if the statistic is consistent with the Galois group being a subgroup of P. If this latter test fails, i.e. $s(R) \not\preceq s(q(e(P)))$, then we remove P from the pool. We also perform the same tests on all maximal subgroups $Q < P \in \mathcal{P}$.

Having processed a resolvent in this way, we may decide to modify $\mathcal{P}$ further, which is controlled by the `Descend` parameter, which is one of the following:

- `Eager`: As soon as there is some $P \in \mathcal{P}$ such that the Galois group is not P, replace P by its maximal subgroups.
- `Steady`: When all $P \in \mathcal{P}$ are known not to be the Galois group, replace the whole pool by the set of maximal subgroups of its elements.

The `Usefulness` algorithm is used by the subgroup choice algorithm to determine whether a potential subgroup $\mathcal{U} \leq \mathcal{W}$ is useful, and so worthy of

consideration. Unlike with the `All` group theory algorithm, where one can always tell whether a subgroup $\mathcal{U}$ will provide any information, in the `Maximal` algorithm we cannot because we do not know all the possibilities of what will occur. Hence we have to use a more heuristic approach and choose one of the following:

- `Necessary`: A subgroup is useful if either: there exist two pool groups $P_1, P_2 \in \mathcal{P}$, both of which might be the Galois group, and whose statistics differ; or there exists a pool group $P \in \mathcal{P}$, which might be the Galois group, and a maximal subgroup $Q < P$, which might contain the Galois group, whose statistics differ.

  These properties are necessary to make any progress with this algorithm, but do not guarantee progress. This is the same as the definition of useful in the `Maximal2` algorithm.

- `Generous`: As with `Necessary`, but Q can be a maximal subgroup of a different pool group.

- `Sufficient`: As with `Generous`, but we require the statistic of P to be inconsistent with being a subgroup of Q, i.e. $s(q(e(P))) \not\trianglerighteq s(q(e(Q)))$.

  A useful group under this definition does not always exist, but when it does the algorithm is guaranteed to make progress.

- `All`: Everything is useful.

We have determined the Galois group when $\mathcal{P}$ contains one group, and we have deduced that the Galois group is not contained in any of its maximal subgroups.

5.3 `Maximal2`

Note that a shortcoming of the `Maximal` algorithm is that it is not always possible to tell if a subgroup $\mathcal{U} \leq \mathcal{W}$ will provide any information, and so its behaviour is more heuristic than principled. Another problem is that it only ever rules groups out of consideration which cannot contain the Galois group, and therefore all groups P with $G \leq P \leq W$ will be considered in the pool $\mathcal{P}$ at some point; if there are many such groups, this can get inefficient. The `Maximal2` algorithm

avoids both of these problems by positively identifying groups which do contain the Galois group.

Parameters.

- A statistic algorithm (§6).
- A subgroup choice algorithm (§7).

As before, we have a pool $\mathcal{P}$ of subgroups, at least one of which contains the Galois group. Suppose there is a group $\mathcal{U} \leq \mathcal{W}$ such that $s(q(e(P))) \not\sim s(q(e(Q)))$ for some $P \in \mathcal{P}$ and maximal $Q < P$ (such a group is **useful**) and we form the corresponding resolvent R. There are two possibilities.

If $s(R) \sim s(q(e(P)))$ then $s(q(e(Q))) \prec s(R)$, so $s(R) \not\preceq s(q(e(Q)))$, so $G \not\leq Q$, and so we can rule Q out of consideration.

Otherwise $s(R) \not\sim s(q(e(P)))$ and so $G \neq P$. In the `Maximal` algorithm at this point we would do something like replace P in the pool by its maximal subgroups. Instead, we find the set X'' of subgroups $Q'' < q(e(P))$ which are maximal among those such that $s(Q'') \sim s(R)$; we refer to these as the **maximal preimages in** $q(e(P))$ **of** $s(R)$. Then we let $X = \{P \cap e^{-1}(q^{-1}(Q'')) : Q'' \in X''\}$. By construction, if $G \leq P$ then $G \leq Q'$ for some $Q' \in X$ and so we can replace P in the pool by X. Typically X is much smaller than the number of maximal subgroups of P.

Suppose now that we have eliminated all maximal subgroups of all $P \in \mathcal{P}$ from consideration. Then we know that $G = P$ for some $P \in \mathcal{P}$. We are now in the scenario of the `All` algorithm, and so can now eliminate groups from the pool by finding $\mathcal{U} \leq \mathcal{W}$ such that $s(q(e(P_1))) \not\sim s(q(e(P_2)))$ for some $P_1, P_2 \in \mathcal{P}$. Such a $\mathcal{U}$ is also said to be **useful**.

We have deduced the Galois group when the pool contains a single group, and we have ruled all of its maximal subgroups out of consideration.

We can use any statistic which has an equivalence relation (as required for `All`) and a partial ordering (as required for `Maximal`) and an algorithm for computing maximal preimages. For the latter, in general we have a "naive" algorithm, which simply works down the subgroups of P until ones with the correct statistic are found.

Algorithm 5.2 (Maximal preimages: Naive)**.** Given a group P, a statistic s and a value v of s, returns the maximal preimages of v in P.

1: **if** $v \sim s(P)$ **then**
2: **return** $\{P\}$
3: **else if** $v \prec s(P)$ **then**
4: **return** $\bigcup_{\text{maximal } Q < P}$ maximal preimages of v in Q
5: **else**
6: **return** $\emptyset$
7: **end if**

Some statistics are stronger than others; for example the `NumRoots` statistic, which counts the number of fixed points of a group, implies the `HasRoot` statistic, which is true if a group has a fixed point. These implications are made explicit in our code. Hence, if one statistic implies a second, and we have an efficient algorithm for finding maximal preimages of the second (this is currently the case for statistics `HasRoot` (§6.1) and `FactorDegrees` (§6.5)), we can use the latter to get closer to the maximal preimages of the former. As a final resort, we use the naive algorithm.

Algorithm 5.3. (Maximal preimages: General) Given a group P, statistic s and a value v of s, this returns the maximal preimages in P of v.

1: $X \leftarrow \{P\}$
2: $S \leftarrow$ statistics implied by s with an efficient algorithm
3: **for** $s' \in S$ **do**
4: $X \leftarrow \bigcup_{Q \in X}$ maximal preimages in Q due to s'
5: **end for**
6: **return** $\bigcup_{Q \in X}$ maximal preimages in Q of v (using naive algorithm)

5.4 `RootsMaximal`

This is essentially the same as `Maximal2` using the `HasRoot` statistic. The pool $\mathcal{P}$ always contains one group P, and the subgroup choice algorithm is to choose any maximal subgroup $Q < P$ which might still contain the Galois group, and use $\mathcal{U} = e(Q)$. This works because of this well-known result:

Lemma 5.4. *$G \leq Q$ if and only if the resolvent R has a root.*

Proof. $G \leq Q$ if and only if $e(G) \leq e(Q)$ (assuming e is injective) if and only if $q(e(G))$ has a fixed point, where $q : \mathcal{W} \to S_{\mathcal{W}/e(Q)}$ is the coset action. Since $\mathrm{Gal}(R) = q(e(G))$, this occurs if and only if R has a root. □

Therefore, if R has a root we can change the pool to $\{Q\}$, and otherwise we can rule Q out from consideration.

Remark 5.5. This is the group theory part of Stauduhar's original algorithm over $\mathbb{Q}$ [65]. It is included more as a demonstration of the generality of our framework than for utility.

5.5 `Sequence`

This takes as parameters a sequence of group theory algorithms. Each one is used in turn until either the Galois group is deduced or the subgroup choice algorithm runs out of subgroups to try.

If the same algorithm appears consecutively with different parameters, then the state of the algorithm (such as the pool of possible Galois groups) is maintained so that information is not lost.

This allows us, for example, to first use a cheap statistic on a limited number of subgroups — aiming to deduce easy Galois groups quickly — before trying a more expensive statistic.

6 Statistic algorithms

A statistic algorithm is a means of comparing the Galois group of a polynomial with a permutation group. Specifically it is a function which takes as input a permutation group or a polynomial and outputs some value. There must be an equivalence relation on these values, which we denote $\sim$. A statistic function s must satisfy the following property: $s(R) \sim s(\mathrm{Gal}(R))$ for all polynomials R. For most statistics, $\sim$ is equality.

Using this, if we are given a polynomial $R(x)$ (such as a resolvent) and a permutation group G and we find that $s(R) \not\sim s(G)$, then we know that $\mathrm{Gal}(R) \neq G$. This is the basis of the `All` (§5.1) group theory algorithm.

Optionally, statistics can also support a partial ordering, denoted $\preceq$, which must respect the partial ordering due to subgroups. Specifically, the following must hold: for all groups G, H, if $H \leq G$ then $s(H) \preceq s(G)$. Statistics supporting this operation may be used in the `Maximal` (§5.2) and `Maximal2` (§5.3) group theory algorithms.

Optionally, ordered statistics can also provide a specialised algorithm to compute maximal preimages, as defined in §5.3.

6.1 `HasRoot`

$s(G)$ is true if it has a fixed point, and otherwise is false. Correspondingly, $s(R)$ is true if it has a root (in its base field K).

If $H \leq G$ and G has a fixed point, then so does H, so we define $v_1 \preceq v_2$ to be $v_2 \implies v_1$.

The maximal subgroups with a fixed point are point stabilizers. Two point stabilizers are conjugate if they stabilize a point in the same orbit, and so we deduce the following algorithm to compute maximal preimages.

Algorithm 6.1. (Maximal preimages: `HasRoot`) Given a group P and a value $v \in \{\text{true}, \text{false}\}$, returns the maximal preimages of v in P.

1: **if** $v = \text{true}$ **then**
2: **return** $\{\mathrm{Stab}_P(x)$ for some $x \in o : o \in \mathrm{Orbits}(P)\}$
3: **else**
4: **return** $\{P\}$
5: **end if**

6.2 `NumRoots`

$s(G)$ is the number of fixed points of G. Correspondingly, $s(R)$ is the number of roots of R.

If $H \leq G$ then H has at least as many fixed points as G, so $\preceq$ in this case is the usual $\leq$ on integers.

6.3 `Factors`

This takes a parameter, which is another statistic s'. Then $s(G)$ is the multiset $\{s'(G')\}$ where G' runs over the images of G acting on each of its orbits (so the degree of G' is the size of the corresponding orbit). Correspondingly, $s(R)$ is the multiset $\{s'(R')\}$ where R' runs over the irreducible factors of R.

6.4 `Degree`

$s(G)$ is the degree of the permutation group G and $s(R)$ is the degree of R.

If $H \leq G$, then they are permutation groups of equal degree, so $v_1 \preceq v_2$ is $v_1 = v_2$.

6.5 `FactorDegrees`

$s(G)$ is the multiset of sizes of orbits of G. Correspondingly, $s(R)$ is the mulitset of degrees of irreducible factors of R.

This is equivalent to `Factors[Degree]` but is more efficient because it does not require the explicit computation of the orbit images of G on its orbits.

Additionally, it supports ordering as follows: we know that if $H \leq G$ then the orbits of H form a refinement of the orbits of G; that is, the orbits of G are unions of orbits of H. Hence, given two multisets v_1 and v_2 of orbits sizes, we check combinatorially if one is a refinement of the other. This is an application of the binning Algorithm 12.5 where the items are v_1, the bins are v_2, and a valid binning $\{d_{1,i}\}$ for bin d_2 has $\sum_i d_{1,i} = d_2$.

We provide an algorithm to compute maximal preimages of this statistic. First, in case the group G is intransitive, we embed G into a direct product D and find maximal preimages there. For each preimage H, and $d \in D$ we see if any $H^d \cap G$ is a preimage. Observing that if $n \in N_D(H)$ and $g \in G$ then $H^{ndg} \cap G = (H^d \cap G)^g$, it suffices to only consider coset representatives of $N_D(H)\backslash D/G$.

Algorithm 6.2 (Maximal preimages: `FactorDegrees`)**.** Given a group G of degree d and a multiset v of integers such that $\sum v = d$, returns all maximal preimages of v in G up to conjugacy.

1: $S \leftarrow \emptyset$
2: Embed $G \subset D = G_1 \times \ldots \times G_r$ (Algorithm 12.1)
3: **for** maximal preimages H of v in D (Algorithm 6.3) **do**
4: **for** double coset representatives d of $N_D(H)\backslash D/G$ **do**
5: $H' \leftarrow H^d \cap G$
6: **if** H' has orbits of sizes v **then**
7: $S \leftarrow S \cup \{H'\}$
8: **end if**
9: **end for**
10: **end for**
11: **return** S

To find maximal preimages in direct products, we first find all the ways in which v may be written as a union, with each component corresponding to a direct factor. Then by Lemma 8.2, the maximal preimages in D are direct products of the maximal preimages in each (transitive) factor.

Algorithm 6.3 (Maximal preimages: `FactorDegrees`: Direct products)**.** Given a direct product $G = G_1 \times \ldots \times G_r$ and v as above, returns all maximal preimages of v in G up to conjugacy.

1: $S \leftarrow \emptyset$
2: **for** multisets $(v_1, \ldots, v_r)$ of integers such that $\sum v_i = \deg G_i$ and $\bigcup_i v_i = v$ **do**
3: **for** i = 1, . . . , r **do**
4: $S_i \leftarrow$ maximal preimages of v_i in G_i (Algorithm 6.4)
5: **end for**
6: **for** $(H_1, \ldots, H_r) \in \prod_i S_i$ **do**
7: $S \leftarrow S \cup \{H_1 \times \ldots \times H_r\}$
8: **end for**
9: **end for**
10: **return** S

To find maximal preimages in transitive groups, we embed G into a wreath

product W, and solve the problem there. As with Algorithm 6.2, a loop over coset representatives lifts these to all preimages in G.

Algorithm 6.4 (Maximal preimages: `FactorDegrees`: Transitive)**.** Given a transitive group G and v as above, returns all maximal preimages of v in G up to conjugacy.

1: $S \leftarrow \emptyset$
2: Embed $G \subset W = G_r \wr \ldots \wr G_1$ (Algorithm 12.2)
3: **for** maximal preimages H of v in W (Algorithm 6.6) **do**
4: **for** double coset representatives w of $N_W(H)\backslash W/G$ **do**
5: $H' \leftarrow H^w \cap G$
6: **if** H' has orbits of sizes v **then**
7: $S \leftarrow S \cup \{H'\}$
8: **end if**
9: **end for**
10: **end for**
11: **return** S

Remark 6.5. Sometimes, if the wreath product W is very large compared to G, the number of double cosets to check makes Algorithm 6.4 infeasible. In this case, we use the naive algorithm instead.

For wreath products, we work recursively so that we only need to consider a single wreath product $A \wr B$. By Lemma 8.4, the maximal preimages correspond to choosing a partition $\mathcal{X}$ for B, and for each $X \in \mathcal{X}$ a partition $\mathcal{Y}_X$ for A, with $v = \{|X|\,|Y| : Y \in \mathcal{Y}_X, X \in \mathcal{X}\}$. We can think of v as the areas of a $d \times e$ rectangle which has a series of vertical cuts (corresponding to the sizes of $\mathcal{X}$), and each piece (X) having a further series of horizontal cuts (corresponding to the sizes of $\mathcal{Y}_X$). We call this a "rectangle division" (see Figure 2). For each such division, we find all possible corresponding partitions of A and B, and take all combinations to construct the partitions for $A \wr B$.

Algorithm 6.6. Given a wreath product $G = W_r \wr \ldots \wr W_1$ and v as above, returns all maximal preimages of v in G up to conjugacy.

1: **if** r = 0 **then**

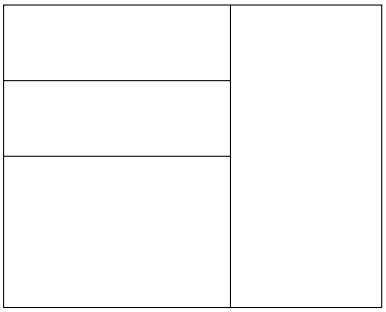

Figure 2: A rectangular division of a 5×4 rectangle, represented as $\{(3, \{2, 1, 1\}), (2, \{4\})\}$, with areas $\{8, 6, 3, 3\}$.

2: **return** $\{G\}$
3: **end if**
4: $A \leftarrow W_r \wr \ldots \wr W_2$
5: $B \leftarrow W_1$
6: $S \leftarrow \emptyset$
7: **for** rectangle divisions $\{(w_i, \{h_{i,j} : j\}) : i\}$ of $\deg A \times \deg B$ into areas v (Algorithm 12.4) **do**
8: $S_B \leftarrow$ maximal preimages of $\{w_i : i\}$ in B (naive Algorithm 5.2)
9: **for** i **do**
10: $S_{A,i} \leftarrow$ maximal preimages of $\{h_{i,j} : j\}$ in A (recursively)
11: **end for**
12: **for** $H_B \in S_B$ **do**
13: $\mathcal{X} \leftarrow \mathrm{Orbits}(H_B)$
14: **for** bijections $m : \mathcal{X} \to \{i\}$ so that $|X| = w_{m(X)}$ **do**
15: **for** $(H_{A,1}, \ldots) \in \prod_i S_{A,i}$ **do**
16: $H \leftarrow \left(\prod_x H_{A,m(\mathcal{X}(x))}\right) \rtimes H_B$
17: $S \leftarrow S \cup \{H\}$
18: **end for**
19: **end for**
20: **end for**
21: **end for**
22: **return** S

We use the naive algorithm to find the maximal preimages of transitive and primitive groups. Since we are mainly dealing with groups close to p-groups, we expect that they have plenty of block structure and therefore the factors in any such wreath product are small enough to use the naive algorithm.

6.6 `NumAuts`

$s(G)$ is the index $(N_G(S) : S)$ where $S := \mathrm{Stab}_G(1)$, assuming G is transitive. $s(R)$ is the number of automorphisms $|\mathrm{Aut}(L/K)|$ where R is irreducible and defines the extension L/K.

Observe that if $G = \mathrm{Gal}(R/K)$, then $S = \mathrm{Gal}(R/L)$, $N_G(S)$ is (by definition) the largest subgroup of G in which S is normal, and hence its fixed field is the smallest subfield M of L/K such that L/M is normal. Hence $\mathrm{Gal}(L/M)$ is $\mathrm{Aut}(L/K)$, and so $\mathrm{Aut}(L/K) \cong N_G(S)/S$.

As we shall see in Lemma 6.7, if $H \leq G$ then $s(G) \mid s(H)$. Hence $v_1 \preceq v_2$ is $v_2 \mid v_1$.

6.7 `AutGroup`

$s(G)$ is the group $N_G(S)/S$ where $S := \mathrm{Stab}_G(1)$ as a regular permutation group of degree $(N_G(S) : S)$; it requires G to be transitive. Correspondingly, $s(R)$ requires R to be irreducible, and is $\mathrm{Aut}(L/K)$ where L is the field defined by R.

$v_1 \sim v_2$ iff v_1 and v_2 are groups of the same degree and are conjugate in the symmetric group of this degree.

The test for ordering uses the following lemma, which says that as the Galois group gets smaller, the automorphism group gets larger. Hence $v_1 \preceq v_2$ is defined as follows: v_1 must have degree at least the degree of v_2, and v_2 must be conjugate to a subgroup of v_1.

Lemma 6.7. *Suppose $G' \leq G$ acts transitvely on a set X. Fix $x \in X$ and define $S := \mathrm{Stab}_G(x)$, $N := N_G(S)$, $A := N/S$ and define S', N', A' similarly with respect to G'. Then A is naturally isomorphic to a subgroup of A'.*

Proof. By definition

$$\begin{aligned}
N &= \{n \in G : s \in S \implies s^n \in S\} \\
&= \{n \in G : s \in S \implies (s^n)(x) = x\} \\
&= \{n \in G : s \in S \implies s(n(x)) = n(x)\} \\
&= \{n \in G : s \in S \implies s \in \mathrm{Stab}_G(n(x))\} \\
&= \{n \in G : S \subseteq \mathrm{Stab}_G(n(x))\} \\
&= \{n \in G : S = \mathrm{Stab}_G(n(x))\} \text{ by orbit-stabilizer theorem} \\
&= \{n \in G : n(x) \in \mathrm{Fix}(S)\} \\
&= \{n \in G : n(y) \in \mathrm{Fix}(S)\} \text{ for any } y \in \mathrm{Fix}(S) \text{ by symmetry} \\
&= \{n \in G : y \in \mathrm{Fix}(S) \implies n(y) \in \mathrm{Fix}(S)\} \\
&= \mathrm{Stab}_G \mathrm{Fix}(S)
\end{aligned}$$

is the group of elements of G which permute the fixed points of $S := \mathrm{Stab}_G(x)$.

Since G is transitive, for each $y \in \mathrm{Fix}(S)$ there exists $n \in G$ such that $n(x) = y$, and hence $n \in N$. We deduce that N acts transitively on $\mathrm{Fix}(S)$, and in particular the orbit-stabilizer theorem implies that

$$|A| = (N : S) = |\mathrm{Fix}(S)| .$$

Similarly, since G' is also transitive then $N \cap G' = \mathrm{Stab}_{G'} \mathrm{Fix}(S)$ acts transitively on $\mathrm{Fix}(S)$, and so the orbit-stabilizer theorem implies

$$|N \cap G'| = |\mathrm{Stab}_{N \cap G'}(1)| \, |\mathrm{Fix}(S)| ,$$

but noting that the stabilizer is actually S' then we deduce

$$(N \cap G' : S') = (N : S).$$

The isomorphism theorems imply

$$(N \cap G')/(S \cap G') \cong (N \cap G')S/S \leq N/S,$$

but noting that $S' = S \cap G'$ then the previous paragraph implies that we have equality, and hence naturally

$$(N \cap G')/(S \cap G') \cong N/S =: A.$$

Finally, note that

$$N \cap G' = \mathrm{Stab}_{G'} \mathrm{Fix}(S) \leq \mathrm{Stab}_{G'} \mathrm{Fix}(S') =: N'$$

so that

$$(N \cap G')/(S \cap G') \leq N'/S' =: A'.$$

□

6.8 `Tup`

This statistic takes as a parameter a tuple $(s_1, \ldots, s_k)$ of statistic algorithms. Then $s(G) = (s_1(G), \ldots, s_k(G))$ and similarly for $s(R)$. Also $v_1 \sim v_2$ iff $v_{1,i} \sim v_{2,i}$ for all i, and similarly for $\preceq$.

6.9 `GalGroup`

$s(G)$ is G itself, and $s(R)$ is $\mathrm{Gal}(R)$, computed using a Galois group algorithm (§2) which is a parameter. Since Galois groups as permutation groups are only defined up to relabelling, then $v_1 \sim v_2$ iff they have the same degree d and are conjugate inside S_d.

6.10 `Order`

$s(G)$ is the order of G. We do not supply a means to compute $s(R)$, and so this statisitic is mainly useful in §11 to dedupe groups up to conjugacy.

7 Subgroup choice algorithms

A subgroup choice algorithm decides, given the current state of a group theory algorithm (§5) for the absolute resolvent method, which subgroup $\mathcal{U} \leq \mathcal{W}$ to form a resolvent from next.

7.1 `Tranche`

Parameters.

- A subgroup tranche algorithm (§8), which produces a sequence $\mathscr{U}_1$, $\mathscr{U}_2$, ... of sets of subgroups of $\mathcal{W}$ one at a time, which we call **tranches**. Given the current tranche, $\mathscr{U}$, we see by inspecting each element in turn if there is $\mathcal{U} \in \mathscr{U}$ such that $\mathcal{U}$ is useful by some measure (see Remark 7.1). If so, we use one such $\mathcal{U}$. If not, we declare the tranche useless and move on to the next one.
- A subgroup priority algorithm (§10), which, given a tranche $\mathscr{U}$, sorts it according to some priority. We then take $\mathcal{U}$ to be the first useful element according to this order. The default is `Null`, which does nothing; empirically, if we try to prioritize so that the most useful groups are considered first, we spend more time prioritizing than we save.

Remark 7.1 (On usefulness). In the `All` group theory algorithm, we have a pool $\mathcal{P}$ of all possible Galois groups, and therefore we know all of the possible outcomes of using the group $\mathcal{U}$ to form a resolvent: i.e. the resolvent has one of the Galois groups $\{q(e(P)) : P \in \mathcal{P}\}$ and so we measure one of the statistic values $\mathcal{S} = \{s(q(e(G))) : P \in \mathcal{P}\}$. If $\mathcal{S}$ contains multiple elements, then $\mathcal{U}$ is useful because we will certainly cut down the list $\mathcal{P}$. Additionally, we define $|\mathcal{S}|$ to be the **diversity of** $\mathcal{U}$ and $\sum_{v \in \mathcal{S}} -p_v \log_2 p_v$ to be the **information in** $\mathcal{U}$, where $p_v = |\{P \in \mathcal{P} : s(q(e(P))) \sim v\}| \,/\, |\mathcal{P}|$ (this is the entropy of the random variable $s(q(e(P))) \in \mathcal{S}$ when $P \sim \mathrm{Unif}(\mathcal{P})$; it may be useful to consider other distributions). These are both measures of usefulness, noting that $\mathcal{U}$ is useful iff its diversity is greater than 1 iff its information is greater than 0, and can be used to prioritize.

In the `Maximal` group theory algorithm, the best definition of useful is more

difficult to establish and is left as a parameter; see §5.2. Usefulness for `Maximal2` is defined in §5.3.

7.2 `Stream`

Parameters.

- A subgroup stream algorithm (§9), which produces a sequence $\mathscr{U}_1$, $\mathscr{U}_2$, ... of possibly infinite sequences of subgroups of $\mathcal{W}$ one at a time, which we call **streams**. The main difference between a stream and a tranche is that the elements of a stream may be generated one at a time, whereas a tranche is generated wholesale. Given the current stream $\mathscr{U}$, we inspect each element in turn to see if there is a $U \in \mathscr{U}$ such that U is useful. We use the first such U. If there is none, we move on to the next stream.
- An integer limit: we only try at most this many items from each stream before moving on to the next one.

8 Subgroup tranche algorithms

A subgroup tranche algorithm takes a permutation group $W \leq S_d$ and returns a sequence of **tranches** $\mathscr{U}_1, \mathscr{U}_2, \ldots$, sets of subgroups of W. The idea is that we want to run through some of the subgroups of W in some order, but that we won't ever actually enumerate all of them, and so it may be more efficient to produce **tranches** according to this order.

8.1 `All`

Produces a single tranche containing all subgroups of W.

8.2 `Index`

For each divisor $n \mid |W|$, produces a tranche containing all the subgroups of W of index n.

Parameters.

- A filtering expression in the free variable `idx` (the index n), which evaluates to true or false. We only produce tranches for the indices n such that the expression is true. By default, we use all n.
- A sorting expression in the free variable `idx`, which evaluates to some sortable value. We sort the indices n according to this expression. By default, we sort by n itself.

There are algorithms to produce the subgroups of a group with a given index. For example, the `Subgroups` intrinsic in Magma has a `IndexEqual` parameter for this purpose.

8.3 `OrbitIndex`

Definition 8.1. For $U \leq W \leq S_d$, the **orbit index of U in W** is the index $(W : U')$ where

$$U' = \mathrm{Stab}_W \mathrm{Orbits}(U) = \{w \in W \,:\, X \in \mathrm{Orbits}(U), x \in X \implies w(x) \in X\}$$

and is denoted $(W : U)^{\mathrm{orb}}$. The **remaining orbit index of U in W** is $(W : U)/(W : U)^{\mathrm{orb}} = (U' : U)$. If $\mathcal{X}$ is a partition of $\{1, \ldots, d\}$, then it is a **subgroup partition for** W if there exists $U \leq W$ such that $\mathcal{X} = \mathrm{Orbits}(U)$. The **index** $(W : \mathcal{X})$ of a subgroup partition $\mathcal{X}$ is $(W : \mathrm{Stab}_W(\mathcal{X}))$.

For each divisor $n \mid |W|$ and $r \mid n$, produces a tranche containing all the subgroups of W of index n and of remaining orbit index r.

Parameters.

- A filtering expression, similar as in `Index` (§8.2). This has free variables `idx` (the index n), `ridx` (the remaining index r) and `oidx` (the orbit index $m := \frac{n}{r}$). By default we use all (n, r) pairs.
- A sorting expression, similar as in `Index`, with the same free variables as the filtering expression. By default, we sort by (n, r) lexicographically (i.e. first by index, then by remaining index).

To produce the tranche corresponding to a given (n, r), we compute the subgroup partitions $\mathcal{X}$ of $\{1, \ldots, d\}$ such that $(W : \mathrm{Stab}_W(\mathcal{X})) = m := \frac{n}{r}$, and

then compute the subgroups of $\mathrm{Stab}_W(\mathcal{X})$ of index r. To efficiently compute the subgroup partitions of W of a given index, we use the special form of W. If W is a wreath product, direct product, or symmetric group, then we can use the algorithms in the rest of this section to reduce the problem to computing subgroup partitions of smaller groups. For these smaller groups, we compute the subgroup partitions by explicitly enumerating all the subgroups.

Lemma 8.2 (Partitions of direct products)**.** *Suppose $W_i \leq S_{d_i}$ for $i = 1, \ldots, k$ (each symmetric group acting on a disjoint set) and $W = W_1 \times \cdots \times W_k$. If $\mathcal{X}_i$ is a partition for W_i of orbit index m_i then $\bigcup_i \mathcal{X}_i$ is a partition for W of orbit index $\prod_i m_i$. Every partition for W is of this form.*

Proof. By definition $m_i = (W_i : \mathrm{Stab}_W(X_i))$. Now

$$\mathrm{Stab}_W(\bigcup_i X_i) = \prod_i \mathrm{Stab}_{W_i}(X_i)$$

and the result follows. Take any $U \leq W$, and consider its projections U_i to W_i, and let $\mathcal{X}_i = \mathrm{Orbits}(U_i)$, then clearly $\mathcal{X} = \bigcup_i \mathcal{X}_i$. □

Algorithm 8.3 (Partitions of direct products)**.** Given $W_i \leq S_{d_i}$ for $i = 1, \ldots, k$ and an integer $m \mid \prod_i |W_i|$, this returns all the partitions for $W = W_1 \times \cdots \times W_k$ of index m.

if $k = 0$ **then**
    return $\{\emptyset\}$
end if
$S \leftarrow \emptyset$
for all $m_1 \mid \gcd(m, |W_1|)$ **do**
    $S_1 \leftarrow$ partitions of W_1 of index m_1
    $S_2 \leftarrow$ partitions of $W_2 \times \cdots \times W_k$ of index $m_2 = \frac{m}{m_1}$
    $S \leftarrow S \cup \{\mathcal{X}_1 \cup \mathcal{X}_2 : \mathcal{X}_1 \in S_1, \mathcal{X}_2 \in S_2\}$
end for
return S

Lemma 8.4 (Partitions of wreath products)**.** *Suppose A, B are permutation groups, let $\mathcal{X}$ be a subgroup partition for B, and for each $X \in \mathcal{X}$ let $\mathcal{Y}_X$ be a*

subgroup partition for A. Then $\mathcal{Z} = \{X \times Y : X \in \mathcal{X}, Y \in \mathcal{Y}_X\}$ is a subgroup partition for $W = A \wr B$, its index is $(B : \mathcal{X}) \prod_{X \in \mathcal{X}} (A : \mathcal{Y}_X)^{|X|}$, and all subgroup partitions are of this form up to conjugacy.

Proof. If A acts on $\{1, \dots, d\}$ and B acts on $\{1, \dots, e\}$, then elements of $A \wr B$ can be defined as elements of the cartesian product $A^e \times B$ acting on $\{1, \dots, e\} \times \{1, \dots, d\}$ as

$$(a_1, \dots, a_e, b)(x, y) = (bx, a_x y).$$

This implies the group operation is

$$(a'_1, \dots, a'_e, b')(a_1, \dots, a_e, b) = (a'_{b1} a_1, \dots, a'_{bd} a_d, b'b).$$

Suppose $\mathcal{Z}$ is defined as above, and take any $(x, y), (x', y') \in X \times Y \in \mathcal{Z}$. Choose $b \in \mathrm{Stab}_B(\mathcal{X})$ such that $b(x) = x'$, which is possible since $\mathrm{Stab}_B(\mathcal{X})$ acts transitively on X by definition of a subgroup partition. Choose $a_x \in \mathrm{Stab}_A(\mathcal{Y}_X)$ such that $a_x(y) = y'$, and choose all other $a_{x''} \in \mathrm{Stab}_A(\mathcal{Y}_{X''})$ for $x'' \in X''$ arbitrarily (e.g. the identity). Defining $g = (a_1, \dots, a_e, b)$ then $g(x, y) = (bx, a_x y) = (x', y')$ and by construction $g \in \mathrm{Stab}_W(\mathcal{Z})$. We conclude that $\mathrm{Stab}_W(\mathcal{Z})$ acts transitively on each element of $\mathcal{Z}$, and so $\mathcal{Z}$ is a subgroup partition of W as claimed.

Expressing $A \wr B$ as a semidirect product $A^e \rtimes B$, then $\mathrm{Stab}_W(\mathcal{Z})$ is the subgroup

$$\left(\prod_{x \in \{1, \dots, e\}} \mathrm{Stab}_A(\mathcal{Y}_{\mathcal{X}(x)}) \right) \rtimes \mathrm{Stab}_B(\mathcal{X})$$

where $\mathcal{X}(x)$ is the $X \in \mathcal{X}$ such that $x \in X$. The index $(W : \mathcal{Z})$ follows.

Suppose $G \leq W$. We want to show that a conjugate of G has orbits of the form $\mathcal{Z}$. Letting $\pi : A \wr B \to B$ be the natural projection $(a_1, \dots, a_e, b) \mapsto b$, let $\mathcal{X} = \mathrm{Orbits}(\pi(G))$, which is a subgroup partition of B. For each $X \in \mathcal{X}$, fix a representative $x_X \in X$, and for each $x \in X$, fix some $g_x = (a_{x,1}, \dots, a_{x,e}, b_x) \in G$ such that $\pi(g_x)(x_X) = x$. Define $\hat{a}_x = a_{x,x_X}$ and $\hat{g} = (\hat{a}_1, \dots, \hat{a}_e, id) \in W$ then by construction

$$g_x^{-1} \hat{g}(x, y) = (x_X, y).$$

Define $\mathcal{Y}_X$ such that $\{x_X\} \times Y$ is an orbit of $S_X := \mathrm{Stab}_G(\{x_X\} \times \{1, \dots, d\})$ for

each $Y \in \mathcal{Y}_X$. We claim that

$$\mathrm{Orbits}(G^{\hat{g}}) = \mathcal{Z} = \{X \times Y \,:\, Y \in \mathcal{Y}_X, X \in \mathcal{X}\}.$$

Note that if $g^{\hat{g}}(x,y) = (x',y')$ then $\pi(g^{\hat{g}})(x) = \pi(g)(x) = x'$ and so $\mathcal{X}(x) = \mathcal{X}(x') = X$ say. For any $(x,y),(x',y')$ with $x,x' \in X \in \mathcal{X}$, then there exists $g \in G$ such that $g^{\hat{g}}(x,y) = (x',y')$ iff there is g such that $(g_{x'}^{-1} g g_x) g_x^{-1} \hat{g}(x,y) = g_{x'}^{-1}\hat{g}(x',y')$, i.e. such that $(g_{x'}^{-1} g g_x)(x_X, y) = (x_X, y')$. This occurs iff there is $g \in S_X$ such that $g(x_X, y) = (x_X, y')$, which occurs iff $\mathcal{Y}(y) = \mathcal{Y}(y') = Y$ say, in which case $(x,y),(x',y') \in X \times Y$. This proves the claim. □

Algorithm 8.5 (Partitions of wreath products)**.** Given $A \leq S_d, B \leq S_e$ and an integer $m \mid |A|^e\,|B|$, this returns all the partitions for $A \wr B$ of index m up to conjugacy.

```
S ← ∅
for all m' | m do
    S' ← partitions for B of index m'
    for all 𝒳 ∈ S' do
        for all factorizations of m/m' of the form ∏_{X∈𝒳} m_X^{|X|} do
            for all X ∈ 𝒳 do
                S_X ← partitions for A of index m_X
            end for
            for all (𝒴_X)_X ∈ ∏_X S_X do
                include {X × Y : X ∈ 𝒳, Y ∈ 𝒴_X} in S
            end for
        end for
    end for
end for
return S
```

Remark 8.6. The preceding algorithm may produce multiple representatives per conjugacy class. With a little more care, we can return just one as follows.

Having chosen $\mathcal{X}$, we partition it into B-conjugacy classes $\mathcal{X}_i = \{X_{i,j}\}$. Then we consider all factorizations of m/m' of the form $\prod_{\mathcal{X}_i} m_i^{|X_{i,1}|}$, and then

all factorizations of m_i of the form $\prod_{X_{i,j} \in \mathcal{X}_i} m_{X_{i,j}}$ with $m_{i,1} \leq m_{i,2} \leq \ldots$. Hence we have a factorization of m/m' of the form $\prod_{X \in \mathcal{X}} m_X^{|X|}$ as above. Note that this includes all factorizations of this form exactly once up to reordering conjugate blocks $X \in \mathcal{X}$.

For such a factorization, we partition $\mathcal{X}_i$ further into classes $\mathcal{X}_{i,j} = \{X_{i,j,k}\}$ such that $m_{i,j} := m_{X_{i,j,k}}$ is constant within a class. Similar to before, we let $S_{i,j} = \{\mathcal{Y}_{i,j,\ell}\}$ be all partitions for A of index $m_{i,j}$, and consider all $(\mathcal{Y}_{i,j,\ell_k})_{i,j,k} \in \prod_{i,j,k} S_{i,j}$ with $\ell_1 \leq \ell_2 \leq \ldots$. Note that this includes all $(\mathcal{Y}_X)_X \in \prod_X S_X$ as above precisely once up to reordering conjugate blocks $X \in \mathcal{X}$.

Letting $\mathcal{Z} = \{X_{i,j} \times Y \,:\, Y \in \mathcal{Y}_{i,j,\ell_k}\}$ be the corresponding partition, then all such $\mathcal{Z}$ are not conjugate in $A \wr B$, and they cover all conjugacy classes up to reordering conjugate blocks of $\mathcal{X}$. Define $S \leq S_d \wr S_e$ to be the group isomorphic to $1_d \wr \prod_i 1_{|\mathcal{X}_i|} \wr S_{|X_{i,1}|}$ which reorders conjugate blocks of $\mathcal{X}$, where 1_d denotes the trivial subgroup of S_d. Then we find all $\mathcal{Z}$ up to $A \wr B$ conjugacy by finding all S-conjugates of $\mathcal{Z}$ up to $A \wr B$ conjugacy as follows.

Let $H_0 = \mathrm{Stab}_{A \wr B}(\mathcal{Z})$, then we want all S-conjugates of H_0 up to $A \wr B$ conjugacy. Note that if $n \in N_S(H_0)$ and $g \in A \wr B$ then $H_0^{nsg} \sim_{A \wr B} H_0^s$ so it suffices to consider double coset representatives s of $N_S(H_0) \backslash S / (A \wr B) \cap S$. Compute H_0^s for all such s and dedupe by $A \wr B$-conjugacy.

Lemma 8.7 (Partitions of symmetric groups)**.** *Any partition $\mathcal{X}$ of $\{1, \ldots, d\}$ is a subgroup partition for S_d and it has orbit index $d!/\prod_{X \in \mathcal{X}} |X|!$.*

Proof. Indeed $\mathrm{Stab}_{S_d}(\mathcal{X}) = \prod_{X \in \mathcal{X}} S_X$. □

Algorithm 8.8 (Partitions of symmetric groups)**.** Given integers $d \geq 0, m \mid d!$, returns all partitions for S_d of index m up to conjugacy.

1: **if** $d = 0$ **then**
2: **return** $\{\emptyset\}$
3: **end if**
4: $S \leftarrow \emptyset$
5: **for all** $d_1 = 0, \ldots, d$ **do**
6: **if** $d!/d_1!(d-d_1)! \mid m$ **then**
7: $S_2 \leftarrow$ partitions of S_{d-d_1} of index $md_1!(d-d_1)!/d!$ up to conjugacy

8: $S \leftarrow S \cup \{\{1, \ldots, d_1\} \cup \mathcal{X}_2 : \mathcal{X}_2 \in S_2\}$
9: **end if**
10: **end for**
11: **return** S

9 Subgroup stream algorithms

Similar to tranche algorithms, a subgroup stream algorithm takes a permutation group $W \leq S_d$ and returns a sequence of **streams** $\mathcal{U}_1, \mathcal{U}_2, \ldots$, which are (possibly infinite) sequences of subgroups of W.

Unlike tranches, which are generated in one go, streams are generated one item at a time, and so are typically used for randomly-generated items.

9.1 `Index`

For each divisor $n \mid |W|$, generates a stream of random subgroups of W of index n.

As with the `Index` tranche algorithm, this takes as parameters filtering and sorting expressions to control which indices n are used and in what order.

We generate a random subgroup of a given index by repeatedly adjoining a random element until the correct index is found. If the correct index is passed, we start over from the trivial subgroup. If we start over too many times (this is a parameter), we give up and assume there is no such group, and therefore terminate the stream.

Algorithm 9.1 (Random subgroup of index)**.** Given a group W and integers n and m, tries m times to generate a random subgroup U of W of index n and returns it, otherwise returns null.

1: **for** $i \in 1, \ldots, m$ **do**
2: $U \leftarrow$ trivial subgroup of W
3: **repeat**
4: $u \leftarrow$ random element of W
5: $U \leftarrow \langle U, u \rangle$.

```
6:        if n ∤ (W : U) then
7:            Go to next i
8:        end if
9:    until (W : U) = n
10:   return U
11: end for
12: return null
```

The algorithm also takes a "dedupe" parameter (§11). If it is not `Null`, then each stream also records the set of conjugacy classes of subgroups it has returned, and only returns subgroups not previously seen. There is a parameter which controls the number of times we have to have generated a subgroup already seen before assuming we have generated them all and therefore terminating the stream.

10 Subgroup priority algorithms

A subgroup priority algorithm takes a tranche $\mathscr{U}$ of subgroups of W and returns an ordering on them.

10.1 `Null`

Does nothing.

10.2 `Random`

Randomizes the order.

10.3 `Reverse`

Takes another priority algorithm as a parameter. Orders according to this, and then reverses it.

10.4 Expression

This is an expression, which is evaluated for each $U \in \mathscr{U}$ and the tranche is sorted by this key.

The expression may have the following free variables:

- `Index`: the index $(W : U)$.
- `OrbitIndex`: the orbit index $(W : U)^{\mathrm{orb}}$ (see Definition 8.1).
- `Diversity`: the diversity (see Remark 7.1).
- `Information`: the information (see Remark 7.1).

11 Deduping algorithms

At various places in our algorithms we have sets of groups which we may want to "dedupe", that is, remove the duplicates up to conjugacy in some larger group. A deduping algorithm takes as input a group W and returns some representation of the set of its conjugacy classes. A subgroup of W can be coerced to its conjugacy class, and a conjugacy class can be hashed so that it can be stored in a hash table. In particular, a set may be implemented as a hash table, and so conjugacy classes can be efficiently deduped.

There are efficient algorithms to test if two subgroups are conjugate (e.g. implemented as the `IsConjugate` intrinsic in Magma) and this is how we test for equality of two conjugacy classes with the same hash.

In fact, deduping tends to take more time than it saves, so we usually use the `None` algorithm, which does not dedupe.

Remark 11.1. In particular, we can dedupe our pool $\mathcal{P}$ of candidate Galois groups in our group theory algorithms. If we do not dedupe, then we can still detect that there is one conjugacy class in $\mathcal{P}$ by choosing some $P \in \mathcal{P}$ and then testing if each other $P' \in \mathcal{P}$ is conjugate to P. This requires only $|\mathcal{P}|$ conjugacy tests, in comparison to $|\mathcal{P}|^2$ tests for a naive deduping algorithm.

11.1 `None`

This does not perform any deduping. That is, each conjugacy class is assigned a distinct hash when generated, so no two classes have the same hash, even if they are equal.

11.2 `Pairwise`

At the other extreme, each conjugacy class is given the same hash. Therefore generating a set of conjugacy classes requires checking if each pair is conjugate.

11.3 `ClassFunc`

Each conjugacy class is given a hash based on some properties of the groups in the class. A parameter controls which hash function to use, and can for example be one of:

- Let $c : W \to \mathbb{Z}$ be a class function for elements of W, i.e. $c(w_1) = c(w_2)$ iff $w_1 \sim_W w_2$. Then the hash for $U \subset W$ is the multiset $\{c(u) : u \in U\}$. A better hash is the multiset of pairs $(c(u), m)$ where u runs over conjugacy classes of elements of U, and m is the size of the class. Computing the class function c itself is prohibitively slow for larger W.
- For transitive permutation groups of small enough degree, its T-number.
- Fix a sequence N_i of normal subgroups of W. Then the hash of $U \subset W$ is the sequence $|W \cap N_i|$.

11.4 `Tree`

We represent the conjugacy classes as a decision tree, whose nodes represent sets of conjugacy classes, and whose leaves are single conjugacy classes. Then a conjugacy class is represented by the corresponding leaf node, which is given a unique integer ID which is its hash. Specifically, this tree has three types of nodes:

- A **leaf** node, which represents a single conjugacy class. These are always the leaves of the tree. Attached to the node is a group representing the class.

- A **decision** node, which has at least two child nodes. Attached to the node is a decision function which takes a group and returns which child node its conjugacy class belongs to.
- An **ambiguous** node, which has at least two child nodes, which are all leaf nodes.

The tree is dynamic, in that it is initially empty and gets populated automatically as conjugacy classes are generated from groups. That is, given a group $G \leq W$, we find its class as follows:

- If the tree is empty, replace it by the tree with a single leaf representing the class of G. Otherwise, traverse the tree starting at the root node as follows.
- If the node is a leaf, representing the class of G', test if G is conjugate to G'. If so, we have found the leaf representing the class of G and so we are done. Otherwise, we have a new class. If we can find a function which distinguishes between the classes of G and G', then replace the leaf node with a decision node with two child leaf nodes for G and G'. Otherwise, replace the leaf node with an ambiguous node with two child leaf nodes. Either way, we have a new leaf representing the class of G and we are done.
- At a decision node, use its decision function on G to traverse to one of its children.
- Otherwise we are at an ambiguous node, in which case check to see if G is conjugate to any of the groups G' representing its child leaves. If so, we have found its class. If not, add a new leaf to the node representing the class of G.

Our decision functions use machinery already developed, namely a subgroup choice algorithm (§7) and a statistic function (§6), which are parameters. Given a pair of groups $G, G' \leq W$ to distinguish, we use the subgroup choice algorithm to search for a subgroup U such that $s(q(G)) \not\sim s(q(G'))$ where s is the statistic and q is the coset action of W on U. Then $s \circ q$ is the decision function. If we run out of groups U to try, then we fail to find a decision function and so fall back on using an ambiguous node instead.

12 Auxillary algorithms

A collection of algorithms used elsewhere in this article.

12.1 Group embeddings

Algorithm 12.1 (Embed into direct product)**.** Given $G \leq S_d$, returns $s \in S_d$ and transitive $G_1, \ldots, G_r$ such that $\sum_i \deg G_i = d$ and $G^s \leq G_1 \times \ldots \times G_r$.

The algorithm finds the image of the group acting on each of its orbits, then takes the direct product.

1: $X_1 = \{x_{1,1}, \ldots\}, \ldots, X_r \leftarrow \text{Orbits}(G)$
2: $D \leftarrow G|_{X_1} \times \ldots \times G|_{X_r}$
3: $s \leftarrow$ permutation sending $x_{i,j}$ to $\sum_{i'<i} |X_{i'}| + j$.
4: **return** D, s^{-1}

Algorithm 12.2 (Embed into wreath product)**.** Given transitive $G \leq S_d$, returns $s \in S_d$ and primitive $G_1, \ldots, G_r$ such that $\prod_i \deg G_i = d$ and $G^s \leq G_r \wr \ldots \wr G_1$.

We choose a minimal non-trivial block-partition of G and use this to embed G into $A \wr B$ with the same block structure. We then recurse on B.

Note that, unlike with the direct product case, there is not a canonical "best" (smallest) choice for the factors. Indeed, suppose we are given a group of the form $(A_1 \times \ldots \times A_e) \rtimes B \leq S_d \wr S_e$, then we could embed it into $A \wr B$ where $A = \langle A_1^{s_1}, \ldots, A_e^{s_e} \rangle$ for any $s_i \in S_d$. Minimizing A is difficult. However, something cheap to compute is: for each $i > 1$, choose $g_i \in G$ such that $g_i(1) \in \{d(i-1)+1, \ldots, di\}$, and then reorder $(d(i-1)+1, \ldots, di)$ to $(g_i(1), \ldots, g_i(d))$ (i.e. let s_i permute $j \mapsto g_i(j) - d(i-1)$). If the best A is cyclic C_d, this is guaranteed to find it.

1: **if** G is primitive **then**
2: **return** G, id
3: **end if**
4: $P \leftarrow$ minimal partition of G
5: Fix an ordering $(B_1, \ldots, B_e)$ on P
6: Fix an ordering $(x_{i,1}, \ldots, x_{i,d})$ on each B_i
7: **for** i = 1,...,e **do**
8: $g_i \leftarrow$ an element of G such that $g_i(x_{1,1}) = g_i(x_{i,1})$

end for
$s \leftarrow$ the permutation $g_i(x_{1,j}) \mapsto di + j$
$G' \leq S_d \wr S_e \leftarrow G^s$
$q : G' \to S_e$ the quotient
$b : S_e \to S_d \wr S_e$ the canonical lift such that $b(\sigma)(id + j) = \sigma(i)d + j$
$B \leftarrow q(G')$
$\mathcal{A} \leftarrow \emptyset$
for each generator g of G **do**
    $g' \leftarrow gb(q(g))^{-1}$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{j \mapsto g'((i-1)d + j) - (i-1)d : i = 1, \ldots, e\} \subset S_d$
end for
$A \leq S_d \leftarrow \langle \mathcal{A} \rangle$
$(W_r, \ldots, W_1), s' \leftarrow$ embedding of B into a wreath product
return $(A, W_r, \ldots, W_1), sb(s')$

12.2 Combinatorial

Algorithm 12.3 (Linear divisions)**.** Given $n \in \mathbb{N}$ and multiset $N \subset \mathbb{N}$, returns all subsets $M \subset N$ such that $\sum M = n$.

We represent multisets of integers as a sorted sequence, with the largest element first. We loop over possible choices of the first division, and then recurse to assign the rest. An additional optional parameter L is such a sequence, and restricts any returned M to be at most L in the lexicographic ordering (i.e. if $M \neq L$, then at the first place they disagree, M must be smaller). The default L is $\{n\}$, which is no restriction.

$S \leftarrow \emptyset$
for distinct $m_1 \in N$ **do**
    if $m_1 \leq \min(n, L_1)$ **then**
        for linear divisions $(m_2, \ldots)$ of $n - m_1$ from $N - \{m_1\}$ with limit $(L_2, \ldots)$ if $m_1 = L_1$ or else limit $(m_1, m_1, \ldots)$ **do**
            Append $(m_1, m_2, \ldots)$ to S
        end for
    end if

8: **end for**
9: **return** S

Algorithm 12.4 (Rectangle divisions)**.** Given $w, h \in \mathbb{Z}$ and multiset $A \subset \mathbb{Z}$, returns all multisets $\{(w_i, \{h_{i,j}\})\}$ such that $\sum_i w_i = w$, $\sum_j h_{i,j} = h$ for each i and $\{w_i h_{i,j} : i, j\} = A$. See Figure 2.

As with the previous algorithm, multisets are represented as sorted sequences. We loop over possible choices of the first division, and then recurse to assign the rest. Optional parameter $L = (w_L, \{h_{L,j}\})$ limits the allowed divisions, with default $(w, \{h\})$.

1: $S \leftarrow \emptyset$
2: **for** distinct divisors w_1 of some $a \in A$ **do**
3: **if** $w_1 \leq \min(w, w_L)$ **then**
4: **for** linear divisions $(h_{1,1}, h_{1,2}, \ldots)$ of h from $\{a/w_1 : a \in A, w_1 \mid a\}$ with limit $h_{L,j}$ if $w_1 = w_L$ or else no limit **do**
5: **for** rectangle divisions $\{(w_2, \{h_{2,j}\}), \ldots\}$ of width $w - w_1$, height h, areas $A - \{w_1 h_{1,j}\}$, limit $(w_1, \{h_{1,j}\})$ **do**
6: Append $(w_1, \{h_{1,j}\}, \ldots)$ to S
7: **end for**
8: **end for**
9: **end if**
10: **end for**
11: **return** S

Algorithm 12.5 (Binning)**.** Suppose we are given integers $(m_1, \ldots, m_r)$ and $(n_1, \ldots, n_s)$ such that we have m_i indistinguishable copies of some item i, and n_j indistinguishable copies of some bin j. A **binning** is some $(m'_1, \ldots, m'_r)$ with $0 \leq m'_i \leq m_i$ for all i. Suppose we are given a function V such that when $V((m'_1, \ldots, m'_r), j)$ is true, we define the binning to be **valid for bin** j. Suppose we are given a function S such that whenever $(m'_1, \ldots, m'_r)$ is valid for bin j and $0 \leq m''_i \leq m'_i$, then $S((m''_1, \ldots, m''_r), j)$ is true. Such a binning is **semi-valid**. A **total valid binning** is a sequence of length s, whose jth entry is a multiset of n_j valid binnings for bin j, and such that all of these binnings sum to $(m_1, \ldots, m_r)$. This algorithm returns all total valid binnings.

Optionally, a **partial semi-valid binning** B can be given (like a total valid binning, except the binnings are only semi-valid and only sum to at most $m_1, \ldots,$ m_r) and this algorithm only returns total valid binnings which extend it.

Optionally, a limit N can be given (defaulting to ∞) and this algorithm returns at most this many total binnings.

This algorithm works by choosing an item and considering all bins it could be added to. For each choice, we add this to the partial semi-valid binning, and recursively find all the total binnings extending it.

In order to avoid duplicated effort, we only add item i to a bin if either it makes the binning equal to the largest or exceed the largest, when comparing the ith entry in each binning.

Since items are assigned in order, they should be given to the algorithm in whatever order is likely to lead to a contradiction quickest (in terms of not being semi-valid). This usually means the "largest" items should come first, because these will "fill" the bins quicker.

(Check semi-valid)
if B is not semi-valid **then**
    return $\emptyset$
end if

(Base case: nothing more to bin)
if $m_i = 0$ for all i **then**
    if B is a total valid binning **then**
        return $\{B\}$
    else
        return $\emptyset$
    end if
end if

(General case)
$\mathcal{R} \leftarrow \emptyset$
$i \leftarrow \min\{i : m_i \neq 0\}$
$m_i \leftarrow m_i - 1$

17: *(Put an item i into one of the j bins)*
18: **for** $j = 1, \ldots, k$ **do**
19: $B = \mathcal{B}_i$ (a set of multiset binnings of size n_j)
20: $B' \leftarrow$ the set of binnings in B with entry i set to 0
21: **for** $b' \in B'$ **do**

22: *(Increase the highest value)*
23: $m_i'' \leftarrow$ the largest value of b_i among all $b \in B$ agreeing with b' away from the ith entry
24: $b_0 \leftarrow b'$ with the ith entry set to m_i''
25: $b \leftarrow b'$ with the ith entry set to $m_i'' + 1$
26: $\mathcal{R} \leftarrow \mathcal{R} \cup$ all total valid binnings extending B with b_0 replaced by b in B_i

27: *(Increase the next one down)*
28: $b_{-1} \leftarrow b'$ with the ith entry set to $m_i'' - 1$
29: **if** $b_{-1} \in B$ **then**
30: $\mathcal{R} \leftarrow \mathcal{R} \cup$ all total valid binnings extending B with b_{-1} replaced by b_0 in B_i
31: **end if**
32: **end for**
33: **end for**
34: **return** $\mathcal{R}$

13 Implementation

These algorithms have been implemented [27] for the Magma computer algebra system [8]. Our main `GaloisGroup` routine takes two arguments: a polynomial over a p-adic field, and a string describing the parameterization of the algorithm to use.

The polynomial itself can be in one of the following three forms:

- The usual "inexact" p-adic polynomial type built in to Magma, which uses capped-precision arithmetic. At the present time, some routines such as

polynomial factoring do not always produce correct results, particularly if the inputs are given to small precision.

- The "exact" p-adic polynomial type made available by the `ExactpAdics` package (see Chapter IV). This uses infinite-precision arithmetic and its routines are designed to give provably correct results (modulo coding errors) and hence our algorithm also yields provably correct results except for Remark 3.6.

- Fixing the Galois group G (and perhaps the ramification filtration and some related data) of some unspecified normal extension L/K, we represent subfields of L/K by the subgroup H of G fixing them, we represent irreducible polynomials over K by the conjugacy class of subgroups $[H]$ of G fixing the fields defined by its roots, and we represent a product of irreducible polynomials by the set of its factors. This allows us to test how well our algorithms can find a given Galois group G without needing an explicit polynomial with this Galois group.

Unless otherwise stated, we use the exact form.

Our algorithm is by design highly modular, with each piece of the parameterization as independent as possible from the rest. This means that if one has a new algorithm for evaluating resolvents for instance, one simply needs to implement this algorithm satisfying a particular interface, and then add a line of code to the parameterization parser.

The main omission from our implementation is that the `SinglyWild` global model algorithm is not fully implemented, which means that for wild extensions our global model will usually use symmetric groups. Over $\mathbb{Q}_2$ with a $2 \times \ldots \times 2$ ramification filtration this is not a problem, but for coarser filtrations, S_8 is much larger than C_2^3 for example, and S_7 is much larger than C_7, and so our global models are far from optimal. A special case of `SinglyWild` has been implemented and is discussed specifically in §13.9.

All experiments reported on in this section were performed on a 2.7GHz Intel Xeon. Any timings are given in core-seconds. Tables of Galois groups have been produced from all runs in this section and are available from the implementation website [27].

13.1 Some particular parameterizations

Six parameterizations we will consider are named A0, B0, A1, B1, A2 and B2 and are all of the form

```
[Tame,
 SinglyRamified,
 ARM[Global:Factors:RamTower:INNER_MODEL, GROUP_ALG]
]
```

where `INNER_MODEL` depends on the letter part of the name, and `GROUP_ALG` depends on the number part.

This parameterization means that we will try three algorithms in turn: `Tame` (§2.2; only works on polynomials whose factors define tame extensions), `SinglyRamified` (§2.3; only works on irreducible polynomials defining singly ramified extensions) and `ARM` (§2.4; absolute resolvent method). The absolute resolvent method evaluates resolvents using a global model which first factorizes the polynomial, then finds the ramification tower of the field defined by each factor, then finds a global model for each segment of the tower using the `INNER_MODEL`.

For the A parameterizations, the inner model is `Symmetric` whereas for the B parameterizations it is

```
Select
  [unram, RootOfUnity[Minimize:True, Complement:True]]
  [tame, RootOfUniformizer]
  [Symmetric:SinglyRamified]
```

which uses `RootOfUnity`, `RootOfUniformizer` or `Symmetric` depending on whether the segment is unramified, tame or wild. In the unramified case, we find a complement to minimize the size of the global model, and in the wild case we use the `SinglyRamified` algorithm to find the Galois group of the segment.

Finally, the `GROUP_ALG` controls the group theory part of the algorithm and depends on the number part of the parameterization name:

0. `All[FactorDegrees]:Index`.

   This writes down `All` possible Galois groups (§5.1), and then tries to

eliminate possibilities based on the `FactorDegrees` statistic (§6.5), which finds the multiset of the degrees of the factors of a polynomial. We consider making resolvents from all groups of each `Index` (§8.2).

1. `All[FactorDegrees]:OrbitIndex[If:le[val[ridx],1]]`.
   Like the previous, but now only makes resolvents from groups such that $v_p(r) \leq 1$ where r is the remaining orbit index (§8.3). Empirically, these groups usually yield as much information as all groups of the same index, although not always: see §13.4.

2. `Maximal2[FactorDegrees]:OrbitIndex[If:le[val[ridx],1]]`.
   Like the previous, but instead of writing down all possible Galois groups, we work down the graph of possible Galois groups using the `Maximal2` algorithm (§5.3).

We shall also consider the following parameterization, called 00.

```
[Tame,
 SinglyRamified,
 ARM[Global:Factors:Symmetric, MaximalRoots]
]
```

It is the same as A0 but uses a `Symmetric` global model for each factor and the `MaximalRoots` group theory algorithm. Hence this parameterization is similar to Stauduhar's original absolute resolvent method [65].

13.2 Up to degree 12 over $\mathbb{Q}_2$, $\mathbb{Q}_3$ and $\mathbb{Q}_5$

The local fields database (LFDB) [40] tabulates data about all extensions of degree up to 12 over $\mathbb{Q}_p$ for all p including a defining polynomial, residue and ramification degrees, Galois and inertia groups, and the Galois slope content which summarizes the ramification polygon of the Galois closure.

We have run our algorithm with the eight paramaterizations `Naive`, 00 and A0 to B2 on all defining polynomials from the LFDB of degrees 2 to 12 over $\mathbb{Q}_2$, $\mathbb{Q}_3$ and $\mathbb{Q}_5$. We also ran with the parameterization A0 but using the inexact polynomial

representation, which we denote A0*. In all cases, the Galois group agrees with that reported in the LFDB.

The mean run times of these are given in Tables 1, 2 and 3. In each case, the times within 10% of the smallest are shown in bold. Counts marked with an asterisk (*) represent a random sample of all possibilities. Times marked with a numeric superscript mean that the algorithm failed to find the Galois group for this many polynomials; these are not included in the mean. A dash (—) means the corresponding algorithm was not tried. A cross (×) means the corresponding runs were prohibitively slow. Times preceded by ≈ are the mean of a small number of runs, the rest being prohibitively slow. This notation is reused in subsequent tables.

Over $\mathbb{Q}_2$, we have also run the algorithm on a selection of reducible polynomials whose irreducible factors have a given set of degrees. For example, we consider all pairs $F_1, F_2 \in K[x]$ of quadratic polynomials defining quadratic fields over $\mathbb{Q}_2$ and run the algorithm on $F(x) = F_1(x)F_2(x+1)$. Note that the offset $x+1$ ensures that $F(x)$ is squarefree in case $F_1 = F_2$. Mean run times are given in Table 1, where for example degree "2 + 2 = 4" means products of quadratics. In all cases, we have verified that the number and sizes of orbits of the returned group is correct, and that the action of the group on each of its orbits agrees with the Galois group reported in the LFDB. Furthermore, in the cases where $F_1 = F_2$, then the Galois group is the diagonal embedding of $\mathrm{Gal}(F_1)$ into $\mathrm{Gal}(F_1) \times \mathrm{Gal}(F_1)$, which again has been verified against the LFDB.

Observe that A0* is generally faster than A0, suggesting there is some overhead due to using exact arithmetic. However, this overhead is around a factor of two in the worst case and usually less, so not too significant.

There is little variation in timings between the six parameterizations A0 to B2. This suggests that for small degrees, there is little overhead in writing down all possible Galois groups $G \leq W$, or in enumerating all subgroups of $\mathcal{W}$ of a given index.

Unsurprisingly, the run time increases in both the degree d and in $v_p(d)$, the latter being the number of wild ramification breaks possible.

Not displayed in the table is that the variance in these run times is low. In particular, the maximum run time is always within a factor of 3 of the mean, and

Degree	#	Run time (seconds)								
		Naive	00	A0*	A0	B0	A1	B1	A2	B2
2	7	**0.03**	0.07	0.04	0.07	0.07	0.07	0.07	0.06	0.07
3	2	**0.08**	0.15	0.10	0.14	0.15	0.16	0.16	0.15	0.15
4	59	**0.05**	0.16	0.09	0.19	0.23	0.19	0.23	0.19	0.23
$2+2=4$	28	—	—	—	**0.20**	0.23	**0.19**	0.22	**0.19**	0.23
5	2	**0.08**	0.15	0.10	0.15	0.15	0.15	0.16	0.15	0.15
6	47	1.32	0.28	**0.13**	0.24	0.28	0.25	0.27	0.26	0.28
$4+2=6$	413	—	—	—	**0.34**	0.39	**0.34**	0.40	**0.35**	0.42
$3+3=6$	3	—	—	—	**0.12**	**0.13**	**0.13**	**0.12**	**0.12**	**0.12**
7	2	**0.09**	0.18	0.12	0.15	0.15	0.15	0.16	0.15	0.15
8	1823	≈ 100	≈ 50	**0.45**	0.59	0.65	0.59	0.65	0.58	0.69
$6+2=8$	329	—	—	—	**0.43**	**0.47**	**0.43**	**0.47**	**0.44**	0.49
$4+4=8$	1770	—	—	—	**0.57**	0.65	**0.58**	0.68	**0.56**	0.80
9	3	0.15	0.15	**0.08**	0.12	0.12	0.12	0.13	0.12	0.12
10	158	≈ 90	$\times$	**0.32**	0.43	0.47	0.44	0.48	0.49	0.48
11	2	0.46	0.17	**0.10**	0.15	0.16	0.15	0.15	0.18	0.17
12	5493	$\times$	$\times$	—	1.26	1.31	**1.18**	**1.21**	**1.11**	1.23
$8+4=12$	1000*	—	—	—	10.97	11.19	10.31	10.25^{1}	**1.33**	1.63
$6+6=12$	1128	—	—	—	2.56	1.74	2.54	1.70	**0.99**	**0.96**
14u	78	$\times$	$\times$	—	1.45	3.97	**0.96**	5.89	4.05	4.62
14t	510	$\times$	$\times$	—	3.05	**1.19**	1.73	**1.19**	1.98	**1.14**
16a	64*	$\times$	$\times$	—	53.65	54.54	17.47^{4}	18.21^{4}	$\mathbf{7.25}^{4}$	$\mathbf{7.59}^{4}$
16b	253*	$\times$	$\times$	—	304.97	288.25	42.37^{7}	34.90^{7}	$\mathbf{25.47}^{7}$	29.40^{7}
16c	130*	$\times$	$\times$	—	$\times$	$\times$	133.29^{23}	195.59^{23}	$\mathbf{115.38}^{4}$	150.83^{23}
18	2046	—	—	—	≈ 100	**1.80**	≈ 75	**1.73**	≈ 35	**1.70**
20	511318	—	—	—	(used several parameterizations; see §13.6)					**10.70**
22	8190	—	—	—	—	**2.90**	—	**2.77**	—	**2.90**

Table 1: Mean run times for some parameterizations on polynomials defining fields of given degrees over $\mathbb{Q}_2$.

Deg	#	Run time (seconds)								
		Naive	00	A0*	A0	B0	A1	B1	A2	B2
2	3	**0.04**	0.11	0.07	0.10	0.11	0.12	0.12	0.11	0.11
3	10	0.05	0.07	**0.04**	0.06	0.06	0.06	0.06	0.05	0.06
4	5	0.10	0.10	**0.05**	0.08	0.08	0.08	0.12	0.09	0.09
5	2	**0.08**	0.16	0.10	0.15	0.16	0.15	0.16	0.14	0.16
6	75	0.66	0.29	**0.13**	0.31	0.33	0.34	0.32	0.30	0.32
7	2	0.12	0.17	**0.10**	0.15	0.18	0.19	0.15	0.16	0.17
8	8	0.10	0.09	**0.06**	0.09	0.08	0.09	0.08	0.08	0.08
9	795	≈ 400	≈ 100	—	**0.63**	**0.64**	**0.67**	**0.66**	**0.66**	0.73
10	6	0.14	0.09	**0.08**	0.09	0.09	0.09	0.10	0.09	0.10
11	2	0.15	0.16	**0.11**	0.17	0.17	0.18	0.19	0.21	0.20
12	785	$\times$	$\times$	—	**1.52**	**1.57**	1.90	2.24	2.21	2.54

Table 2: Mean run times for some parameterizations on polynomials defining fields of given degrees over $\mathbb{Q}_3$. There were 11 polynomials of degree 12 for which A0, A1 and A2 did not succeed due to a bug in Magma; these are not included in timings.

Deg	#	Run time (seconds)								
		Naive	00	A0*	A0	B0	A1	B1	A2	B2
2	3	**0.04**	0.11	0.07	0.12	0.28	0.11	0.11	0.12	0.11
3	2	**0.09**	0.14	**0.10**	0.15	0.15	0.15	0.15	0.20	0.16
4	7	**0.03**	0.07	0.04	0.07	0.07	0.08	0.07	0.09	0.08
5	26	0.12	0.05	**0.02**	0.05	0.06	0.05	0.06	0.05	0.06
6	7	0.07	0.09	**0.05**	0.08	0.08	0.08	0.09	0.08	0.08
7	2	0.12	0.17	**0.10**	0.15	0.16	0.16	0.16	0.15	0.21
8	11	0.07	0.09	**0.05**	0.08	0.07	0.08	0.08	0.07	0.09
9	3	0.12	0.11	**0.09**	0.15	0.13	0.13	0.13	0.13	0.13
10	258	≈ 100	$\times$	—	**2.09**	**1.93**	3.00	2.76	16.02	11.87
11	2	0.15	0.17	**0.11**	0.18	0.17	0.17	0.19	0.18	0.44
12	17	0.16	**0.08**	**0.09**	**0.08**	**0.08**	**0.08**	**0.08**	**0.08**	**0.08**

Table 3: Mean run times for some parameterizations on polynomials defining fields of given degrees over $\mathbb{Q}_5$.

is usually less.

For small degrees, the simple parameterization 00 is comparable to the other parameterizations. However it quickly becomes infeasible as the degree increases, taking for example about 50 seconds at degree 8 over $\mathbb{Q}_2$.

The same is true for the `Naive` algorithm. Indeed, for small degrees this is often the fastest but becomes infeasibly slow above degree about 10.

13.3 Degree 14 over $\mathbb{Q}_2$

There are two types of wildly ramified extensions $L/K = \mathbb{Q}_2$ of degree 14: those with $e(L/K) = 2$ and those with $e(L/K) = 14$. In the former case, L is a ramified quadratic extension of the unique unramified extension U/K of degree 7. In the latter case, L is a ramified quadratic extension of the unique (tamely) ramified extension $T = K(\sqrt[7]{2})/K$ of degree 7. We refer to these as Type 14u and Type 14t respectively.

Using the `AllExtensions` intrinsic in Magma we have generated all such extensions. There are 510 of each type up to U- or T-conjugacy, and after deduping by K-conjugacy there are 78 of Type 14u and 510 of Type 14t (these were already distinct because $\mathrm{Aut}(T/K)$ is trivial).

We have run our algorithm on all of these. The timings are given in Table 1 separately for the two types.

As a point of comparison, [3] uses a degree 364 resolvent relative to $W = S_{14}$ and a few other invariants to compute the same Galois groups, taking around 20 hours per polynomial whereas our algorithm takes around 2 seconds. Our results are consistent with [3, Table 3].

We see that for Type 14t, using a more sophisticated global model `RootOfUniformizer` for T/K in the B parameterizations instead of `Symmetric` in the A parameterizations makes a marked improvement to the run-time. Even when we do use `Symmetric`, we get an improvement for using more sophisticated group theory, comparing A0, A1 and A2.

In contrast, for Type 14u using a more sophisticated global model `RootOfUnity` actually made the run time worse. In this case, with parameterization B0, most of the run time is spent computing complex approximations to resolvents, despite

generally using fewer resolvents and using a lower complex precision. This suggests that the implementation of `RootOfUnity` needs to be optimized.

We have verified that the order of the Galois group is correct using class field theory as follows. Let $M = U$ for Type 14u and $M = T$ for Type 16t and let N be its normal closure, so $N = U$ or $N = T(\zeta_7)$. Let $G_N = \mathrm{Gal}(N/K) = C_7$ or $C_7 \rtimes C_3$. Now since L/M is quadratic and wildly ramified, so is LN/N. Compute the unit group $A = N^\times$ and norm group $B = N_{LN/N}(LM^\times) \leq A$ and the intersection $C = \bigcap_{g \in G_N} g(B) \leq A$. The class field of C is the normal closure of L/K and therefore in particular $(A : C)(N : K) = |\mathrm{Gal}(L/K)|$. We have checked that this relationship holds in all cases.

Note that up to S_{14}-conjugacy, there is only one transitive subgroup of $C_2 \wr C_7$ or $C_2 \wr (C_7 \rtimes C_3)$ of each possible order. Hence verifying the order of the Galois group is the same as verifying the group itself up to S_{14}-conjugacy.

13.4 Degree 16 over $\mathbb{Q}_2$

Recall (e.g. [53] or Chapter III) that to an extension of p-adic fields, we can attach a ramification polygon, which is an invariant of the extension. By attaching further residual information such as the residual polynomials of each face of the ramification polygon, we can form a finer invariant.

Using the `pAdicExtensions` package [26], which implements these invariants, we generated all possible equivalence classes of the finest such invariant, called the **fine ramification polygon with residues and uniformizer residue** in Chapter III, for totally ramified extensions of degree 16 of $\mathbb{Q}_2$.

For each class, we selected at random one Eisenstein polynomial generating a field with this invariant, giving us a sample of 447 polynomials.

We divide these polynomials into three types. Writing $L = L_t/\ldots/L_0 = K = \mathbb{Q}_2$ for the ramification filtration of the field they generate, then Type 16a polynomials have $(L_i : L_{i-1}) = 2$ for all i (and hence $t = 4$), Type 16b polynomials are those remaining with $(L_i : L_{i-1}) \mid 4$ for all i, and Type 16c are the rest (so $(L_i : L_{i-1}) = 8$ or 16 for some i). There are 64, 253 and 130 polynomials of each type respectively.

In total, there are 4,008,960 degree 16 extensions of $\mathbb{Q}_2$ inside $\bar{\mathbb{Q}}_2$ of Type 16a,

	A0	B0	A1	B1	A2	B2
Type 16a (64 polynomials)						
Number failed	0	0	4	4	4	4
Mean run time	53.65	54.54	17.47	18.21	7.25	7.59
Median run time	27.87	28.64	16.69	17.00	6.06	6.34
Maximum run time	311.86	252.39	31.57	56.59	22.99	21.76
Type 16b (253 polynomials)						
Number failed	0	0	7	7	7	7
Mean run time	304.97	288.25	42.37	34.90	25.47	29.40
Median run time	18.20	14.77	12.25	10.38	8.02	7.65
Maximum run time	4016.19	3721.84	432.85	1182.44	1063.16	1616.56
Type 16c (130 polynomials)						
Number failed	—	—	23	23	4	23
Mean run time	—	—	133.29	195.59	115.38	150.83
Median run time	—	—	10.50	1.58	1.43	1.36
Maximum run time	—	—	2502.06	7949.19	12432.12	4368.25

Table 4: Run times in seconds for a selection of parameterizations on a sample of polynomials defining fields of degree 16 over $\mathbb{Q}_2$ divided into three types.

1,857,120 of Type 16b and 155,024 of Type 16c [63].

Per an earlier remark, we do not have `SinglyWild` global models implemented and so use the less efficient `Symmetric` instead. We expect run times for Types 16b and 16c to be worse than Type 16a, since the former will work relative to groups like $W = S_4 \wr S_4$ or $S_2 \wr S_8$ which are larger than $W = S_2 \wr S_2 \wr S_2 \wr S_2$ of the latter. We expect that with `SinglyWild` fully implemented, the overgroup for Types 16b or 16c will be smaller not larger than for Type 16a, and that Types 16b and 16c will therefore actually become the easier classes. See §13.9 for some evidence supporting this claim.

Our algorithm has been run on these polynomials with the 6 parameterizations A0 to B2. Table 4 summarizes the results, with the polynomials grouped by type. Mean timings are also given in Table 1 for comparison. Some of these runs failed to find the Galois group, because the parameterization ran out of resolvents to try; the number of failures is given in the table. The timings only include successful runs. To give an idea of the variance in run time, we report the median and maximum time as well as the mean.

The run times are significantly higher at degree 16 than lower degrees, and there are now pronounced differences between the parameterizations, with those numbered 0 being the slowest and numbered 2 being the fastest.

As predicted, Type 16a polynomials are the fastest. For this type, the median is usually close to the mean and the maximum is not much larger, indicating this is a low-variance regime. Elsewhere, the median is smaller and the maximum is a lot higher, so the variance is greater.

Among Type 16a, four polynomials failed on parameterizations A1 to B2. They fail because forming resolvents from groups whose remaining orbit index is at most 2 is not sufficient to distinguish these Galois groups. Our parameterizations A1 to B2 could be modified to fall back on the A0 or B0 behaviour if they run out of groups to try, thus getting most of the efficiency of the more sophisticated parameterizations, but retaining the generality of A0. These polynomials are

$$f_1 = x^{16} + 4x^{14} + 12x^{12} + 2x^8 + 8x^7 + 8x^5 + 4x^4 + 8x^2 + 2$$
$$f_2 = x^{16} + 4x^{14} + 4x^{12} + 8x^{11} + 2x^8 + 16x^7 + 8x^6 + 16x^5 + 4x^4 + 2$$
$$f_3 = x^{16} + 16x^{15} + 8x^{14} + 16x^{13} + 4x^{12} + 2x^8 + 16x^7 + 16x^5 + 4x^4 + 8x^2 + 16x + 6$$
$$f_4 = x^{16} + 16x^{15} + 4x^{12} + 8x^{10} + 16x^9 + 2x^8 + 16x^7 + 16x^5 + 4x^4 + 16x^3 + 40x^2 + 32x + 6.$$

Under these parameterizations, the Galois group of f_1 and f_2 is deduced to be one of 16T896 or 16T920, of order 512. The Galois group of f_3 is deduced to be 16T294 or 16T334, of order 128. The Galois group of f_4 is deduced to be 16T970 or 16T1000, of order 512. The correct Galois groups, computed using A0, are 16T896, 16T920, 16T334 and 16T1000 respectively.

The results are consistent in the sense that for each polynomial, the same group was found for each of the six parameterizations.

Among Type 16a, the smallest Galois groups found were 16T151 and 16T177 of order $64 = 2^6$, and the largest are 16T1704, 16T1709, 16T1712, 16T1727, 16T1739, 16T1741, 16T1742, 16T1743 and 16T1744 of order $8192 = 2^{13}$. The smallest cases have been verified using the `Naive` algorithm.

J	#	Groups
1	2	433, 434
3	4	98, 101, 588, 592
5	8	433^2, 434^2, 588^2, 592^2
7	16	433^4, 434^4, 588^4, 592^4
9	32	45^2, 147^2, 512^{14}, 656^{14}
11	64	433^8, 434^8, 512^{16}, 588^8, 592^8, 656^{16}
13	128	433^{16}, 434^{16}, 512^{32}, 588^{16}, 592^{16}, 656^{32}
15	256	98^2, 101^2, 147^4, 588^{62}, 592^{62}, 656^{124}
17	512	433^{32}, 434^{32}, 512^{64}, 588^{96}, 592^{96}, 656^{192}
18	1024	45^4, 147^{12}, 512^{252}, 656^{756}
Total	2046	45^6, 98^3, 101^3, 147^{18}, 433^{63}, 434^{63}, 512^{378}, 588^{189}, 592^{189}, 656^{1134}

Table 5: Totally ramified Galois groups of degree 18 over $\mathbb{Q}_2$.

13.5 Degree 18 over $\mathbb{Q}_2$

Using the `pAdicExtensions` package [26], we have generated all ramification polygons of totally ramified extensions $L/\mathbb{Q}_2$ of degree 18. These have vertices of the form

$$(1, J), (2, 0), (18, 0)$$

where the discriminant valuation is $18 + J - 1$. Note that these extensions are of the form $L/T/\mathbb{Q}_2$ where $T/\mathbb{Q}_2$ is the unique tame extension of degree 9 and L/T is quadratic.

For each polygon, we have generated a set of polynomials generating all extensions with this ramification polygon, and run our algorithm on them all with parameterizations A0 to B2. There are 2046 polynomials in total.

Mean timings are given in Table 1. Note that the B parameterizations are far quicker than A as a result of using the `RootOfUniformizer` global model instead of `Symmetric` for $T/\mathbb{Q}_2$.

In Table 5 we give the number of polynomials for each ramification polygon (parameterized by J) and the count of the T-numbers of their Galois groups.

The smallest Galois group is 18T45 $\cong C_2 \times (C_9 \rtimes C_6)$ and the largest is 18T656 $= C_2 \wr (C_9 \rtimes C_6)$. Note that these are the smallest and largest possible using only

that $\mathrm{Gal}(T/\mathbb{Q}_2) = C_9 \rtimes C_6$ and $\mathrm{Gal}(L/T) = C_2$.

Noting that L/T is Galois and $T/\mathbb{Q}_2$ has only the trivial automorphism, then $\mathrm{Aut}(L/\mathbb{Q}_2) \cong C_2$ and so each $L/\mathbb{Q}_2$ has 9 conjugates inside $\bar{\mathbb{Q}}_2$. The number of polynomials generated times 9 is equal to the number of extensions of degree 18 in $\bar{\mathbb{Q}}_2$, from which we deduce we have exactly one polynomial per isomorphism class.

13.6 Degree 20 over $\mathbb{Q}_2$

As in §13.5, we have generated all ramification polygons of totally ramified extensions $L/\mathbb{Q}_2$ of degree 20. These have vertices

$$(1, J_0), (2, J_1), (4, 0), (20, 0) \quad \text{or} \quad (1, J_0), (4, 0), (20, 0)$$

and the discriminant has valuation $20 + J_0 - 1$. For each we have produced a set of generating polynomials, 511,318 in total. We have computed the Galois groups of all of these polynomials, an account of which is given later in this section.

By [48, Theorem 1] there are 259,968 isomorphism classes of such extensions $L/\mathbb{Q}_2$ so we have over-counted by a factor of about 2.

The counts of Galois groups are given in Table 6. Note than when there is no vertex $(2, J_1)$ in the ramification polygon there are two cases: either there is a point $(2, J_1)$ in the *fine* ramification polygon (cf. Chapter III) with $J_1 = \frac{2}{3}J_0$ and we prefix J_1 with "="; or there is no such point and we write $J_1 = *$.

Finding the Galois groups. We ran our algorithm with parameterization B2 on these polynomials with a time limit of 15 minutes per polynomial. All but 1700 polynomials succeeded. This took an average of 10.0 seconds per polynomial, about half of this time spent on the failed cases.

We then ran with parameterization B5-500-1000 (described below) on the remainder with a time limit of 5 minutes per polynomial. All but 217 polynomials succeeded, taking an average of 225.5 seconds per polynomial, about 30% of this time spent on the failed cases.

These remaining polynomials all have a global model of the form $\mathcal{W} = S_4 \wr F_5$, that is they define a field $L/T/\mathbb{Q}_2$ where $T = \mathbb{Q}_2(\sqrt[5]{2})/\mathbb{Q}_2$ is tame degree 5 and L/T is singly ramified degree 4. These fall into two types: Type A has

J_0	J_1	#	Groups
1	$*$	1	173
3	$=2$	1	471
3	$*$	2	77, 80
5	2	8	305, 309^2, 312^2, 332^2, 351
5	$*$	2	61, 282
7	2	16	305, 306^3, 309^2, 312^2, 317^2, 330, 332^2, 338^2, 351
7	$*$	4	173, 282, 472, 683
9	2	32	847^8, 850^8, 851^8, 854^8
9	$=6$	4	471^4
9	$*$	8	77^2, 80^2, 317^2, 338^2
11	2	64	129^2, 131^2, 132^2, 137^2, 406^2, 422^2, 434^4, 443^4, 444^2, 447^4, 448^4, 449^2, 847^8, 850^8, 851^8, 854^8
11	6	32	847^8, 850^8, 851^8, 854^8
11	$*$	8	173^2, 282^2, 472^2, 683^2
13	2	128	417^8, 435^8, 437^8, 441^8, 520^{16}, 530^{16}, 908^{32}, 910^{32}
13	6	64	129^4, 131^4, 132^4, 137^4, 406^4, 422^4, 444^4, 449^4, 847^8, 850^8, 851^8, 854^8
13	$*$	16	173^4, 282^4, 472^4, 683^4
15	2	256	305^2, 306^6, 309^4, 312^4, 317^4, 330^2, 332^4, 338^4, 351^2, 406^4, 417^8, 422^4, 434^4, 435^8, 437^8, 441^8, 443^4, 444^4, 447^4, 448^4, 449^4, 520^{16}, 530^{16}, 847^{16}, 850^{16}, 851^{16}, 854^{16}, 908^{32}, 910^{32}
15	6	128	305^4, 309^8, 312^8, 332^8, 351^4, 434^8, 443^8, 447^8, 448^8, 847^{16}, 850^{16}, 851^{16}, 854^{16}
15	$=10$	16	68, 678^{15}
15	$*$	32	19^2, 194^{12}, 195^6, 526^{12}
17	2	512	847^{64}, 850^{64}, 851^{64}, 854^{64}, 908^{128}, 910^{128}

Table 6: (continued overleaf)

J_0	J_1	#	Groups
17	6	256	305^{4}, 306^{12}, 309^{8}, 312^{8}, 317^{8}, 330^{4}, 332^{8}, 338^{8}, 351^{4}, 406^{8}, 422^{8}, 434^{8}, 443^{8}, 444^{8}, 447^{8}, 448^{8}, 449^{8}, 847^{32}, 850^{32}, 851^{32}, 854^{32}
17	10	128	411^{8}, 416^{8}, 419^{8}, 420^{8}, 512^{16}, 514^{16}, 907^{32}, 911^{32}
17	$*$	32	173^{2}, 282^{2}, 472^{14}, 683^{14}
19	2	1,024	305^{4}, 306^{12}, 309^{8}, 312^{8}, 317^{8}, 330^{4}, 332^{8}, 338^{8}, 351^{4}, 406^{8}, 417^{16}, 422^{8}, 434^{8}, 435^{16}, 437^{16}, 441^{16}, 443^{8}, 444^{8}, 447^{8}, 448^{8}, 449^{8}, 520^{32}, 530^{32}, 847^{96}, 850^{96}, 851^{96}, 854^{96}, 908^{192}, 910^{192}
19	6	512	417^{32}, 435^{32}, 437^{32}, 441^{32}, 520^{64}, 530^{64}, 908^{128}, 910^{128}
19	10	256	131^{8}, 137^{8}, 195^{16}, 406^{24}, 411^{8}, 416^{8}, 419^{8}, 420^{8}, 422^{24}, 512^{16}, 514^{16}, 526^{48}, 907^{32}, 911^{32}
19	$*$	64	173^{4}, 282^{4}, 472^{28}, 683^{28}
21	2	2,048	129^{4}, 131^{4}, 132^{4}, 137^{4}, 406^{28}, 422^{28}, 434^{32}, 443^{32}, 444^{28}, 447^{32}, 448^{32}, 449^{28}, 515^{64}, 516^{64}, 520^{64}, 530^{64}, 847^{192}, 850^{192}, 851^{192}, 854^{192}, 908^{384}, 910^{384}
21	6	1,024	305^{8}, 306^{24}, 309^{16}, 312^{16}, 317^{16}, 330^{8}, 332^{16}, 338^{16}, 351^{8}, 406^{16}, 417^{32}, 422^{16}, 434^{16}, 435^{32}, 437^{32}, 441^{32}, 443^{16}, 444^{16}, 447^{16}, 448^{16}, 449^{16}, 520^{64}, 530^{64}, 847^{64}, 850^{64}, 851^{64}, 854^{64}, 908^{128}, 910^{128}
21	10	512	411^{32}, 416^{32}, 419^{32}, 420^{32}, 512^{64}, 514^{64}, 907^{128}, 911^{128}
21	$=14$	64	471^{16}, 678^{48}
21	$*$	128	77^{4}, 80^{4}, 129^{4}, 132^{4}, 194^{16}, 317^{12}, 338^{12}, 444^{12}, 449^{12}, 526^{48}
22	2	4,096	417^{64}, 435^{64}, 437^{64}, 441^{64}, 515^{128}, 516^{128}, 908^{1536}, 910^{2048}
23	6	2,048	129^{8}, 131^{8}, 132^{8}, 137^{8}, 406^{56}, 422^{56}, 434^{64}, 443^{64}, 444^{56}, 447^{64}, 448^{64}, 449^{56}, 515^{128}, 516^{128}, 520^{128}, 530^{128}, 847^{128}, 850^{128}, 851^{128}, 854^{128}, 908^{256}, 910^{256}
23	10	1,024	131^{16}, 137^{16}, 195^{32}, 406^{48}, 411^{48}, 416^{48}, 419^{48}, 420^{48}, 422^{48}, 512^{96}, 514^{96}, 526^{96}, 907^{192}, 911^{192}

Table 6: (continued overleaf)

J_0	J_1	#	Groups
23	14	512	847^{64}, 850^{64}, 851^{64}, 854^{64}, 907^{128}, 911^{128}
23	$*$	128	173^{8}, 282^{8}, 472^{56}, 683^{56}
25	6	4,096	847^{512}, 850^{512}, 851^{512}, 854^{512}, 908^{1024}, 910^{1024}
25	10	2,048	16^{4}, 19^{4}, 194^{24}, 195^{60}, 196^{84}, 511^{192}, 512^{192}, 514^{192}, 518^{192}, 526^{168}, 528^{168}, 907^{384}, 911^{384}
25	14	1,024	306^{48}, 317^{32}, 330^{16}, 338^{32}, 406^{96}, 422^{96}, 444^{32}, 449^{32}, 526^{128}, 847^{64}, 850^{64}, 851^{64}, 854^{64}, 907^{128}, 911^{128}
25	$*$	256	61^{2}, 282^{44}, 683^{210}
26	6	8,192	417^{128}, 435^{128}, 437^{128}, 441^{128}, 515^{256}, 516^{256}, 908^{3072}, 910^{4096}
27	10	4,096	416^{128}, 420^{128}, 511^{256}, 514^{512}, 906^{1024}, 907^{512}, 909^{1024}, 911^{512}
27	14	2,048	129^{16}, 131^{16}, 132^{16}, 137^{16}, 194^{32}, 196^{32}, 406^{112}, 422^{112}, 444^{112}, 449^{112}, 526^{224}, 528^{224}, 847^{128}, 850^{128}, 851^{128}, 854^{128}, 907^{256}, 911^{256}
27	$=18$	256	471^{64}, 678^{192}
27	$*$	512	77^{8}, 80^{8}, 129^{8}, 132^{8}, 194^{32}, 317^{56}, 338^{56}, 444^{56}, 449^{56}, 526^{224}
29	10	8,192	416^{256}, 420^{256}, 511^{512}, 514^{1024}, 906^{2048}, 907^{1024}, 909^{2048}, 911^{1024}
29	14	4,096	435^{128}, 441^{128}, 516^{256}, 530^{512}, 906^{1024}, 908^{512}, 909^{1024}, 910^{512}
29	18	2,048	129^{32}, 131^{32}, 132^{32}, 137^{32}, 194^{64}, 196^{64}, 406^{224}, 422^{224}, 444^{224}, 449^{224}, 526^{448}, 528^{448}
29	$*$	512	173^{8}, 282^{24}, 472^{120}, 683^{360}
30	10	16,384	42^{16}, 261^{240}, 632^{768}, 633^{3840}, 634^{768}, 946^{10752}
31	14	8,192	305^{32}, 309^{64}, 312^{64}, 332^{64}, 351^{32}, 411^{128}, 417^{128}, 419^{128}, 434^{192}, 437^{128}, 443^{192}, 447^{192}, 448^{192}, 512^{512}, 515^{256}, 518^{256}, 520^{512}, 847^{256}, 850^{256}, 851^{256}, 854^{256}, 906^{1024}, 907^{512}, 908^{512}, 909^{1024}, 910^{512}, 911^{512}

Table 6: (continued overleaf)

J_0	J_1	#	Groups
31	18	4,096	847^{512}, 850^{512}, 851^{512}, 854^{512}, 907^{1024}, 911^{1024}
31	20	4,096	411^{64}, 416^{64}, 419^{64}, 420^{64}, 511^{128}, 518^{128}, 907^{2048}, 911^{1536}
33	14	16,384	847^{1024}, 850^{1024}, 851^{1024}, 854^{1024}, 906^{2048}, 907^{2048}, 908^{2048}, 909^{2048}, 910^{2048}, 911^{2048}
33	18	8,192	435^{256}, 441^{256}, 516^{512}, 530^{1024}, 906^{2048}, 908^{1024}, 909^{2048}, 910^{1024}
33	20	8,192	411^{128}, 416^{128}, 419^{128}, 420^{128}, 511^{256}, 518^{256}, 907^{4096}, 911^{3072}
34	14	32,768	417^{256}, 435^{256}, 437^{256}, 441^{256}, 515^{512}, 516^{512}, 632^{1024}, 634^{1024}, 908^{6144}, 910^{8192}, 946^{14336}
35	18	16,384	847^{1024}, 850^{1024}, 851^{1024}, 854^{1024}, 906^{2048}, 907^{2048}, 908^{2048}, 909^{2048}, 910^{2048}, 911^{2048}
35	20	16,384	42^{16}, 261^{240}, 632^{3840}, 633^{768}, 634^{768}, 946^{10752}
37	18	32,768	305^{64}, 309^{128}, 312^{128}, 332^{128}, 351^{64}, 411^{256}, 417^{256}, 419^{256}, 434^{384}, 437^{256}, 443^{384}, 447^{384}, 448^{384}, 512^{1024}, 515^{512}, 518^{512}, 520^{1024}, 847^{1536}, 850^{1536}, 851^{1536}, 854^{1536}, 906^{4096}, 907^{3072}, 908^{3072}, 909^{4096}, 910^{3072}, 911^{3072}
37	20	32,768	411^{256}, 416^{256}, 419^{256}, 420^{256}, 511^{512}, 518^{512}, 633^{1024}, 634^{1024}, 907^{8192}, 911^{6144}, 946^{14336}
38	18	65,536	417^{512}, 435^{512}, 437^{512}, 441^{512}, 515^{1024}, 516^{1024}, 632^{2048}, 634^{2048}, 908^{12288}, 910^{16384}, 946^{28672}
39	20	65,536	411^{512}, 416^{512}, 419^{512}, 420^{512}, 511^{1024}, 518^{1024}, 633^{2048}, 634^{2048}, 907^{16384}, 911^{12288}, 946^{28672}
40	20	131,072	18^{8}, 20^{8}, 42^{16}, 186^{120}, 189^{120}, 261^{240}, 510^{1920}, 517^{1920}, 519^{1920}, 523^{1920}, 524^{1920}, 529^{1920}, 632^{3840}, 633^{3840}, 634^{3840}, 906^{26880}, 909^{26880}, 946^{53760}

Table 6: (continued overleaf)

J_0	J_1	#	Groups
	Total	511,318	16^{4}, 18^{8}, 19^{6}, 20^{8}, 42^{48}, 61^{3}, 68, 77^{15}, 80^{15}, 129^{78}, 131^{90}, 132^{78}, 137^{90}, 173^{30}, 186^{120}, 189^{120}, 194^{180}, 195^{114}, 196^{180}, 261^{720}, 282^{90}, 305^{120}, 306^{105}, 309^{240}, 312^{240}, 317^{140}, 330^{35}, 332^{240}, 338^{140}, 351^{120}, 406^{630}, 411^{1440}, 416^{1440}, 417^{1440}, 419^{1440}, 420^{1440}, 422^{630}, 434^{720}, 435^{1440}, 437^{1440}, 441^{1440}, 443^{720}, 444^{562}, 447^{720}, 448^{720}, 449^{562}, 471^{85}, 472^{225}, 510^{1920}, 511^{2880}, 512^{1920}, 514^{1920}, 515^{2880}, 516^{2880}, 517^{1920}, 518^{2880}, 519^{1920}, 520^{1920}, 523^{1920}, 524^{1920}, 526^{1396}, 528^{840}, 529^{1920}, 530^{1920}, 632^{11520}, 633^{11520}, 634^{11520}, 678^{255}, 683^{675}, 847^{5760}, 850^{5760}, 851^{5760}, 854^{5760}, 906^{42240}, 907^{42240}, 908^{34560}, 909^{42240}, 910^{42240}, 911^{34560}, 946^{161280}

Table 6: Totally ramified Galois groups of degree 20 over $\mathbb{Q}_2$.

$\mathrm{Gal}(L/T) = C_2 \wr C_2$, there are 199 of these; and Type B has $\mathrm{Gal}(L/T) = S_4$, there are 18 of these.

We ran with parameterization B5-500-1000-D4 (described below) on the Type A polynomials, which all succeeded. This took an average of 161.60 seconds per polynomial.

The 18 Type B polynomials were the most difficult. Using a parameterization involving the statistic `Factors[Tup[Degree,NumAuts]]` (which on polynomials is the multiset over factors of the tuple of the degree of the factor and the number of automorphisms of the field it defines) we were able to find the Galois group of 9 of these polynomials, but this took about $2\frac{1}{2}$ hours per polynomial.

Instead, we know from [32, Theorem 7.3] that the normal closure of L/T is tamely ramified over L. In particular, if we let $K = \mathbb{Q}_2(\sqrt[3]{2}, \zeta_3)$ then $K/\mathbb{Q}_2$ is Galois with Galois group S_3 and LK/T is the normal closure of L/T, and hence $\mathrm{Gal}(LK/TK) \cong C_2^2$. Note also that $(LK : K) = 20$ and hence F is irreducible over K and so the splitting field of F over $\mathbb{Q}_2$ equals the splitting field of F over K equals the normal closure of LK/K. We use the parameterization C2 introduced in §13.9

to find $G_0 := \mathrm{Gal}(F/K) = \mathrm{Gal}(LK/K)$. Then if $G := \mathrm{Gal}(F/\mathbb{Q}_2) = \mathrm{Gal}(L/\mathbb{Q}_2)$ then we know that $G_0 \triangleleft G \leq S_4 \wr S_5$, such that $G/G_0 \cong \mathrm{Gal}(K/\mathbb{Q}_2) \cong S_3$ and the image of G under the quotient $S_4 \wr S_5 \to S_5$ is $\mathrm{Gal}(T/K) \cong F_5$. For each G_0, we compute that there is a unique group G with these properties (either 20T282 or 20T683, 9 of each), and hence we have found the Galois group. Computing G_0 took an average of 66.51 seconds per polynomial. About 80% of this time was spent constructing the global model, which is now much slower since we are working over an extension $K/\mathbb{Q}_2$.

Parameterization B5-500-1000.

```
[Tame,
 SinglyRamified,
 ARM[Global:Factors:RamTower:Select
     [unram, RootOfUnity[Minimize:True, Complement:True]]
     [tame, RootOfUniformizer]
     [Symmetric:SinglyRamified],
  [All[FactorDegrees, OrbitIndex[If:and[le[val[ridx],1],le[idx,500]]]],
   All[FactorDegrees, Stream[1000, Index]]
  ]
 ]
]
```

This is the same as B2 except for the group theory part. Whereas in B2 we use the `Maximal2` group theory algorithm and `OrbitIndex` to choose groups, this parameterization uses the `All` algorithm (it does not take long to enumerate all possible subgroups in our application) and initially chooses groups using `OrbitIndex` up to index 500, and then uses `Stream[1000,Index]` to try up to 1000 random subgroup of each index.

Parameterization B5-500-1000-D4 This is the same as B5-500-1000 except that the global model construction is modified. `Symmetric:SinglyRamified` is replaced by `D4Tower` so that in our application for Type A polynomials we get $W = \mathcal{W} = C_2 \wr C_2 \wr F_5$ instead of $S_4 \wr F_5$.

J	#	Groups
1	2	34, 35
3	4	34^2, 35^2
5	8	34^4, 35^4
7	16	34^8, 35^8
9	32	34^{16}, 35^{16}
11	64	6^2, 37^{62}
13	128	34^{32}, 35^{32}, 37^{64}
15	256	34^{64}, 35^{64}, 37^{128}
17	512	34^{128}, 35^{128}, 37^{256}
19	1024	34^{256}, 35^{256}, 37^{512}
21	2048	34^{512}, 35^{512}, 37^{1024}
22	4096	6^4, 37^{4092}
Total	8190	6^6, 34^{1023}, 35^{1023}, 37^{6138}

Table 7: Totally ramified Galois groups of degree 22 over $\mathbb{Q}_2$.

13.7 Degree 22 over $\mathbb{Q}_2$

As in §13.5, we have generated all ramification polygons of totally ramified extensions $L/\mathbb{Q}_2$ of degree 22, these have vertices of the form

$$(1, J), (2, 0), (22, 0),$$

and for each we have produced a set of generating polynomials. Again, we have precisely one polynomial per isomorphism class, 8190 in total.

Timings with parameterizations B0 to B2 are given in Table 1 and counts of Galois groups are given in Table 7.

The Galois groups range from 22T6 $\cong C_2 \times F_{11}$ to 22T37 $= C_2 \wr F_{11}$, where $F_{11} = C_{11} \rtimes C_{10} = \mathrm{Gal}(T/\mathbb{Q}_2)$ is the Galois group of the unique tame extension $T/\mathbb{Q}_2$ of degree 11.

13.8 Degree 32 over $\mathbb{Q}_2$

A degree 16 extension of $\mathbb{Q}_2$ typically has a ramification filtration of the form $2 \times 2 \times 2 \times 2$, so that we typically find its Galois group as a subgroup of $W = C_2^{\wr 4} = C_2 \wr C_2 \wr C_2 \wr C_2$. Now $|W| = 2^{15}$ and the Galois group has order

at least $16 = 2^4$, so the Galois group is index at most 2^{11} in W. For a typical extension of degree 32, the Galois group is a subgroup of $W = C_2^{\wr 5}$ of index up to $2^{31-5} = 2^{26}$. Since determining $G \leq W$ using the absolute resolvent method tends to be more difficult as the index $(W : G)$ increases, we expect that computing the Galois group of a degree 32 polynomial is in general far more difficult than at degree 16.

Consider

$$f_1(x) = x^{16} + 2x^8 + 16x^5 + 32x^3 + 10$$

which is Eisenstein, defining an extension with a $2\times2\times2\times2$ ramification filtration, and whose Galois group is 16T1722 of index $4 = 2^2$ in $C_2^{\wr 4}$.

Now consider $f_1(x^2)$, which is also Eisenstein and defines an extension with a $2 \times \ldots \times 2$ ramification filtration. Using the A2 parameterization, we can compute its Galois group to be 32T2752023 of index 2^7 in $C_2^{\wr 5}$. This took about 116 seconds, which breaks down as follows.

	Run time (seconds)	Share of run time
Start resolvent algorithm	34.81	29.9%
Choose subgroup	68.99	59.3%
Compute resolvent	2.12	1.8%
Process resolvent	8.25	7.1%
Other	2.25	1.9%
Total	116.42	

Here, "start resolvent algorithm" includes initially factorizing the polynomial, finding the extensions defined by the factors, finding their ramification filtrations, and computing a corresponding global model. "Choose subgroup" means time spent by the subgroup choice algorithm choosing a subgroup $\mathcal{U} \leq \mathcal{W}$ from which to form a resolvent. "Compute resolvent" is the time spent computing a resolvent $R(x)$ given an invariant for the subgroup $\mathcal{U}$. "Process resolvent" is the time spent by the group theory algorithm deducing information about the Galois group from a resolvent, and so in particular includes finding the degrees of the factors of the resolvent and computing maximal preimages. "Other" is everything else, including intializing the group theory algorithm and computing invariants.

This used 82 resolvents in total: 68 of degree 2, 3 of degree 4, 7 of degree 8 and 4 of degree 32. The maximum complex precision used was 2735 decimal digits.

Note that the A0 or A1 parameterizations are prohibitively slow on this problem: there are millions of transitive subgroups of $C_2^{\wr 5}$ to enumerate (using Magma's database of transitive groups, there are 2,737,535 S_{32}-conjugacy classes of transitive 2-groups of degree 32). Even if we did this once and re-used the list of subgroups, computing $s(q(e(P)))$ for every possible P to determine if $\mathcal{U}$ is useful would be extremely time consuming.

Similarly, define

$$f_2(x) = x^{16} + 32x + 2$$

which is Eisenstein with Galois group 16T1638 of index $8 = 2^3$ in $C_2^{\wr 4}$. Using A2, we find the Galois group of $f_2(x^2)$ is 32T2583443 of index 2^{10} in $C_2^{\wr 5}$. This took about 125 seconds, which breaks down as follows.

	Run time (seconds)	Share of run time
Start resolvent algorithm	23.28	18.6%
Choose subgroup	91.44	73.0%
Compute resolvent	1.39	1.1%
Process resolvent	6.84	5.5%
Other	2.37	1.9%
Total	125.32	

This used 104 resolvents in total: 82 of degree 2, 9 of degree 4, 7 of degree 8, 2 of degree 16 and 4 of degree 32. The maximum complex precision used was 4056 decimal digits.

In both cases, the run time is dominated by time spent choosing subgroups $\mathcal{U} \leq \mathcal{W}$, suggesting that this should be the focus for future improvement. The next most dominant part is time spent starting the resolvent algorithm, but this part is essentially independent of the Galois group. Very little time is spent actually computing resolvents, which is perhaps surprising given that this is the part spent using complex embeddings of global models.

13.9 A special case of `SinglyWild`

We have implemented `SinglyWild` in the special case $p = 2$ for totally wildly ramified extensions L/K which are Galois. Hence $\mathrm{Gal}(L/K) \cong C_2^k$ where $(L : K) = p^k$.

We now define three more parameterizations C0, C1 and C2 which are the same as B0, B1 and B2 except that the `Symmetric` global model is replaced by `SinglyWild`. That is, `INNER_MODEL` is

```
Select
  [unram, RootOfUnity[Minimize:True, Complement:True]]
  [tame, RootOfUniformizer]
  [SinglyWild].
```

It is well-known (e.g. [60, Ch. IV, §2, Prop. 7]) that for such an extension L/K there is an injective group homomorphism $\mathrm{Gal}(L/K) \to \mathbb{F}_K^+$, and hence $\mathrm{Gal}(L/K)$ is isomorphic to a subspace of $\mathbb{F}_K/\mathbb{F}_p$. In particular, $(\mathbb{F}_K : \mathbb{F}_p) \geq k$ and so $K/\mathbb{Q}_p$ has residue degree at least k.

Using the `pAdicExtensions` package [26], we have generated defining polynomials which between them generate all extensions of the form $L/U/\mathbb{Q}_2$ where $U/\mathbb{Q}_2$ is unramified of some degree and L/U is singly wildly ramified and Galois of some degree.

For example when $k = 2$ and $(U : \mathbb{Q}_2) = 4$, then the global model in C0 gives the overgroup $W = C_2^2 \wr C_4$ of order 2^{10}, which is somewhat smaller than the overgroup $W = S_4 \wr C_4$ of order $2^{14} \cdot 3^4$ from B0.

We have run our algorithm with the 9 parameterizations A0 to C2 on these polynomials. Mean timings are given in Table 8.

Except at degree 8, the C parameterizations are by far the quickest.

14 Future work

14.1 Improvements

Here are some avenues for improvement to the absolute resolvent method. Note that thanks to the modular design of the implementation, it is quite

Deg	k	#	Run time (seconds)								
			A0	B0	C0	A1	B1	C1	A2	B2	C2
8	2	4	**0.53**	**0.56**	0.61	**0.57**	**0.57**	0.66	0.62	**0.58**	0.68
12	2	28	2.36	2.34	0.71	3.41	3.73	**0.64**	4.57	4.79	**0.66**
16	2	140	×	×	1.23	80.02[1]	23.27[1]	**0.95**	×	×	**0.98**
24	3	8	×	×	**12.55**	×	×	**12.75**	×	×	**12.42**
32	3	120	—	—	40.49	—	—	31.34	—	—	**23.68**

Table 8: Mean run times for a selection of parameterizations on polynomials defining fields of the form $L/U/\mathbb{Q}_2$ where $U/\mathbb{Q}_2$ is unramified and L/U is singly wildly ramified with Galois group C_2^k. At degree 32, there were four polynomials which did not succeed due to a bug in Magma; these are not included in the mean.

straightforward to make improvements to one aspect of the algorithm in isolation from the rest. The items at the top of the list should yield the best improvements.

- The quality of the global model and embedding $e : W \to \mathcal{W}$ determines the number and degree of resolvents required to deduce the Galois group. Hence, it is worth putting in effort to make W and $\mathcal{W}$ as small as possible. We can ask: given an extension L/K of local fields, then among all global models $\mathcal{L}/\mathcal{K}$ how small can $\mathrm{Gal}(\mathcal{L}/\mathcal{K})$ get in comparison to $\mathrm{Gal}(L/K)$? And can we compute $\mathcal{L}/\mathcal{K}$? The first question is answered for abelian L/K by class field theory (see Remark 4.3).

- On larger examples, the run time is currently dominated by time spent computing subgroups $\mathcal{U} \leq \mathcal{W}$ from which to form resolvents. Our current approaches involve constructing a list of candidate $\mathcal{U}$ independent of the state of the algorithm and then testing to see which of these are useful. A better approach would be to construct $\mathcal{U}$ directly.

  As an example of this, suppose $e : W \to W$ is the identity, then $G \leq U$ if and only if the corresponding resolvent has a root in K. Hence if $G_1, G_2 \leq W$ are two potential Galois groups and without loss of generality $G_1 \not\leq G_2$ then choosing $U = G_1$ will distinguish between these two groups. This is the principle behind the `RootsMaximal` algorithm but has the disadvantage that if $(W : G)$ is large, then $(W : U)$ will also be large at some point.

Hence we ask: given $G_1 \neq G_2$ and some statistic algorithm s, then among all U such that a resolvent for U distinguishes between G_1 and G_2 with respect to s, which has the smallest index $(W : U)$? And can we compute U?

A heuristic approach could be to start with $U = G_1$ and keep adjoining random elements of W, only adjoining an element if the resulting U is still useful.

- In §13.6, to find the hardest 18 Galois groups we found it was easier to first compute the Galois group over some small Galois extension known to lie in the splitting field. This helped because we could find a much more compact global model. This strategy could be formalized and built in to our algorithm.

- In this article we have defined many statistics, but `FactorDegrees` turned out to be most useful. This is largely because there are good algorithms for factorizing p-adic polynomials, and also because we have a quick algorithm for computing maximal preimages of this statistic. Other statistics either do not contain as much information (e.g. `HasRoot`) or are much slower to compute (e.g. `AutGroup`). Are there statistics which are fast to evaluate and for which we can quickly compute maximal preimages?

- Similarly, can we get a better algorithm by using a relatively cheap statistic initially, and then swapping to a more expensive one after a certain amount of effort?

- The subfields of the extension L/K correspond to subgroups $H \leq G$ of the Galois group $G = \mathrm{Gal}(L/K)$ containing $\mathrm{Stab}_G(1)$. In turn, these correspond to block partitions of G. Using ideas from [37], we could compute the subfields of L/K, from which we get a corresponding directed acyclic graph whose nodes correspond to subfields, the nodes being labelled with the degree of the field, and with edges corresponding to inclusion. Given a resolvent R defining L/K, this DAG is a statistic for R. The corresponding statistic on groups is the DAG of block partitions.

14.2 Ramification filtration

The core tool at the centre of the absolute resolvent method is being able to compute resolvents, a resolvent being nothing more than a polynomial defining a certain subfield of the splitting field of the original polynomial $F(x) \in K[x]$. By measuring ramification information from resolvents, we can also deduce the inertia group $G_0 \leq G = \mathrm{Gal}(F/K)$ and indeed the whole ramification filtration $(G_0, 0), (G_1, u_1), \ldots, (G_t, u_t)$ where u_i are the ramification breaks and $G_i \triangleleft G$ are the corresponding ramification groups.

By way of example, one way to compute the inertia group is as follows. First, compute the Galois group G. Now find all normal subgroups $G_0 \triangleleft G$ such that G/G_0 is cyclic, and $G_1 \triangleleft G$ and G_0/G_1 cyclic where G_1 is a Sylow p-subgroup of G_0. These are candidate inertia groups. Now if R is a resolvent for $\mathcal{U} \leq \mathcal{W}$, then $\mathrm{Gal}(R/K) \cong q(e(G))$ and $\mathrm{Gal}(R/K)_0 \cong q(e(G_0))$ (the isomorphism is the same) where $q : \mathcal{W} \to S_{\mathcal{W}/\mathcal{U}}$ is the coset action. Hence by measuring information about $\mathrm{Gal}(R/K)_0$ (such as the ramification degree of the extensions defined by the factors of R) we can eliminate candidate G_0.

The main challenge in extending this to computing the full ramification filtration will be in finding the ramification breaks u_i. Using that the upper numbering of ramification breaks is consistent with respect to a fixed ground field K, then the upper ramification breaks of any extension defined by a factor of a resolvent R are a subset of the upper ramification breaks of G. Hence, by choosing the right resolvents we will eventually see all upper breaks.

Of course, in analogue with our algorithms to compute the Galois group itself, we could avoid computing all possible inertia groups G_0 by instead working down the graph of normal subgroups of G.

Chapter III

Generating Extensions

Foreword

This chapter has been published separately as an article [25].

When testing our algorithm for computing Galois groups, it is convenient to have a supply of interesting example polynomials. Pauli and Sinclair [53] give an algorithm to produce Eisenstein polynomials generating extensions with a given ramification polygon and residual polynomials, but no public implementation. We have an implementation [26].

In this chapter, we present their results again using simplified notation which we think yields more illuminating proofs of the main results. The only parts which are new are the algorithms to enumerate all possible (equivalence classes of) invariants (§2.4, §5.2) and the concept of weak validity (§2.3) which makes these enumeration algorithms practical.

As noted in §7, the invariants implemented in our package are actually more general than those described in this chapter or in [53] (but they are weaker, not stronger). Again, this is mainly to aid the enumeration of all possibilities.

1 Introduction

We fix a p-adic field K and an integer $n \geq 1$ and consider the problem of enumerating all the extensions L/K of degree n. In their paper [53], Pauli and

Sinclair describe some invariants of L/K and give an algorithm to produce all extensions with a given set of invariants. The algorithm works by producing a set of Eisenstein polynomials generating all extensions for a given invariant, and then determining which pairs of polynomials produce isomorphic extensions so that only one extension per isomorphism class is returned. The finer the invariant used, the smaller the set of polynomials produced, and so easier this pairwise search becomes.

In §2–6 we give a brief re-exposition of the invariants they describe, using slightly different notation which we hope is easier to follow. For each invariant, we give its definition, prove that it is an invariant, determine which Eisenstein polynomials $f(x)$ generate an extension with the given invariant, and give an algorithm to enumerate all possiblities for the invariant.

In §2 we look at the ramification polygon. This is not studied directly in [53], which actually starts with a slightly finer invariant. However the ramification polygon is more widely known, which is why we start with it. In §3 we look at the "fine ramification polygon" which is the finer invariant studied in [53, §3], in which it is called the "ramification polygon".

In §4 we look at the "fine ramification polygon with residues" which attaches a residue to each point in the polygon. This is equivalent to a pair of a ramification polygon and the invariant $\mathcal{A}$ of [53, §4], but is notationally simpler: $\mathcal{A}$ is expressed as a set of residual polynomials, but this structure is actually mostly irrelevant to studying them.

In §5 we extend this invariant to include some information about the constant coefficient of the Eisenstein $f(x)$. This is essentially the object $\mathcal{A}^*$ of [53, Eq. 4.2], although it is not explicitly identified as an invariant.

In §6 we consider, as in [53, §5], transformations to the Eisenstein polynomial $f(x)$ which preserve the extension L/K, allowing us to reduce the number of such polynomials we need to enumerate in order to find all extensions.

We reiterate that the main results in this article are not new, and are all essentially from [53]. Exceptions to this are made in the footnotes. We do however use different notation and present the proofs in a different manner.

Many of the ideas here were studied by Monge [47] to produce, given a totally ramified extension L/K, a finite set of Eisenstein polynomials each generating

L/K, forming an invariant for L/K. The same ideas were also used by Sinclair to count the number of extensions with a given invariant [63], the count being the number of extensions within an algebraic closure. Monge has also counted the number of extensions of a given degree up to isomorphism [48] using class field theory to count cyclic extensions, and a group theory argument to count conjugacy classes of subgroups.

The invariants and algorithms described here have been implemented [26] in the Magma computer algebra system [8]. In §7 we give a few notes on the implementation. For convenience, our package also provides an implementation of [63, Lemma 4.4] to count the number of extensions L/K in some algebraic closure $\bar{K}$ with a given invariant.

1.1 Notation

K is a finite extension of $\mathbb{Q}_p$. Fix a uniformizer $\pi \in K$, and let v denote the valuation on $\bar{K}$ such that $v(\pi) = 1$. We denote by $\mathcal{O}_K$ the ring of integers of K, $\mathbb{F}_K = \mathcal{O}_K/(\pi)$ the residue class field, and for $x \in \mathcal{O}_K$ we let $\bar{x}$ be its residue class.

For $x, y \in \mathcal{O}_{\bar{K}}$, we say $x \equiv y$ iff $v(x-y) > 0$. For $x, y \in \bar{K}$, we say $x \sim y$ iff $x = y = 0$ or $v(x-y) > v(x) = v(y)$.

2 Ramification polygon

2.1 Definition

For a monic Eisenstein polynomial $f(x) = \sum_{i=0}^{n} f_i x^i \in K[x]$ of degree n, with a root $\alpha \in \bar{K}$, its **ramification polynomial** is

$$r(x) := \alpha^{-n} f(\alpha x + \alpha) \in L[x], \qquad L := K(\alpha).$$

Note that it is monic and $r(0) = 0$. Expanding out we find

$$r(x) = \sum_{j=1}^{n} r_j x^j \quad \text{where} \quad r_j = \sum_{i=j}^{n} \binom{i}{j} f_i \alpha^{i-n} \text{ for } 0 < j \leq n.$$

Observing that $nv(\binom{i}{j} f_i \alpha^{i-n}) \equiv i \mod n$, by the ultrametric property of the valuation we deduce that

$$R_j := nv(r_j) = \min_{i=j}^{n} n(B(i,j) + F_i - 1) + i,$$

where $B(i,j) := v\binom{i}{j}$ and $F_i := v(f_i)$.

We define the **ramification points of** f to be

$$\mathcal{P} = \{(j, R_j) \,:\, 1 \leq j \leq n, R_j < \infty\}$$

and we define the **ramification polygon of** f to be the lower convex hull of these points, denoted P. That is, it is the Newton polygon of $r(x)$. Note that R_j, and hence $\mathcal{P}$, can be computed directly from f without explicitly computing r.

2.2 Invariant

It is well-known that the ramification polygon P is actually an invariant of the field L/K. One way to see this is to observe that if $\hat{\alpha}$ is any uniformizer for L/K, then $\hat{\alpha} = x_0 + x_1\alpha + \ldots + x_{n-1}\alpha^{n-1}$ with $x_i \in \mathcal{O}_K$, $v(x_0) > 0$, $v(x_1) = 0$ (since $\mathcal{O}_L = \mathcal{O}_K[\alpha]$), so for any K-embedding $\sigma : L \to \bar{K}$,

$$\begin{aligned}
\hat{\alpha} - \sigma\hat{\alpha} &= \sum_{i=0}^{n-1} x_i(\alpha^i - \sigma\alpha^i) \\
&= (\alpha - \sigma\alpha)(x_1 + \sum_{i=2}^{n-1} x_i(\alpha^{i-1} + \ldots + \sigma\alpha^{i-1})) \\
&\sim x_1(\alpha - \sigma\alpha)
\end{aligned}$$

and therefore

$$\frac{\hat{\alpha} - \sigma\hat{\alpha}}{\hat{\alpha}} \sim \frac{\alpha - \sigma\alpha}{\alpha}$$

and in particular these have the same valuation. Since these, over all σ, are the roots of the ramification polynomials corresponding to $\hat{\alpha}$ and α, and they have the same valuations, then the Newton polygons of the ramification polynomials are the same.

It is related to the ramification filtration of $\mathrm{Gal}(L/K)$ if L/K is Galois (e.g.

[60, Ch. IV]), and more generally to the ramification filtration of the **Galois set** $\Gamma(L/K) = \{\sigma : L \to \bar{K}\}$ of K-embeddings of L (e.g. [35]). The vertices of the polygon correspond to subfields of L/K which themselves correspond to fixed fields of elements of the ramification filtration of $\Gamma(L/K)$:

$$\Gamma_V := \{\sigma \in \Gamma \,:\, x \in \mathcal{O}_K \implies v(\sigma x - x) > V\} = \{\sigma \in \Gamma \,:\, v(\sigma\alpha - \alpha) > V\}.$$

2.3 Validity

From facts about ramification, or alternatively directly from facts about valuations of binomial coefficients, one can show that the interior vertices of P are of the form $(p^s, *)$, and that there is a vertex at $(p^{v_p(n)}, 0)$. Hence we notate a ramification polygon by listing its vertices like so:

$$P = [(p^{s_0}, J_0), \ldots, (p^{s_u}, J_u = 0), (n, 0)]$$

where $s_0 = 0$ and $s_u = v_p(n)$.

Any polygon of the above form is called a **potential ramification polygon**. Any such polygon arising from an Eisenstein $f(x) \in K[x]$ is called a **valid ramification polygon (over K)**. We now consider which potential ramification polygons are valid.

Observing that P is the graph of a function $[1, n] \to \mathbb{Q}$, we let P also denote this function.

A potential polygon P is valid if and only if there is Eisenstein $f(x)$ with $v(f_i) = F_i$ such that $R_{p^{s_t}} = J_t$ for $0 \le t \le u$ and such that $R_j \ge P(j)$ for all $1 \le j \le n$. Automatically we have $R_j \ge 0$ for all j, which rules the face $[(p^{s_u}, 0), (n, 0)]$ out of consideration. From facts about valuations of binomials, we actually have that if $p^s < j < p^{s+1}$ then $R_j \ge R_{p^s} \ge P(p^s) > P(j)$, and hence the only points (j, R_j) lying in P with $1 \le j \le p^{s_u}$ are of the form (p^s, R_{p^s}). We deduce that P is valid if and only if $R_{p^s} \ge P(p^s)$ for all s with equality whenever $s = s_t$.

Fix some s, then $R_{p^s} \ge P(p^s)$ if and only if for all $p^s \le i \le n$ we have

$$n(B(i, p^s) + F_i - 1) + i \ge P(p^s)$$

which may be rewritten as

$$F_i \geq \ell_P(i,s) := \left\lceil \frac{P(p^s)-i}{n} \right\rceil - B(i,p^s) + 1.$$

Suppose this is true for $s = s_t$, then we have equality $R_{p^{s_t}} = P(p^{s_t}) = J_t =: a_t n + b_t$ with $1 \leq b_t \leq n$[1] if and only if we have equality for $i = b_t$, i.e. $p^{s_t} \leq b_t \leq n$ and

$$F_{b_t} = \ell_P(b_t, s_t) = a_t - B(b_t, p^{s_t}) + 1.$$

We deduce that P is valid if and only if $p^{s_t} \leq b_t \leq n$ and there are $F_0, F_1, \ldots, F_n \in \mathbb{Z}$ satisfying:

$$\begin{aligned} & F_0 = 1 \\ 0 < i < n \implies & F_i \geq 1 \\ & F_n = 0 \\ p^s \leq i \leq n \implies & F_i \geq \ell_P(i,s) \\ & F_{b_t} = \ell_P(b_t, s_t). \end{aligned}$$

Each condition is either a lower bound or an equality for some F_i. Hence this system is consistent if and only if the equalities for the same F_i match, and if the lower bounds are satisfied by the equalities. Hence P is valid if and only if[2]

$$\begin{aligned} & p^{s_t} \leq b_t & \\ b_t = n \implies & \ell_P(n, s_t) = 0 & \text{(Ore 1)} \\ & \ell_P(n,s) \leq 0 & \text{(Ore 2)} \\ b_t < n \implies & \ell_P(b_t, s_t) \geq 1 & \text{(Ore 3)} \\ b_r = b_t < n \implies & \ell_P(b_t, s_t) = \ell_P(b_r, s_r) & \text{(Consistency)} \\ p^s \leq b_t < n \implies & \ell_P(b_t, s_t) \geq \ell_P(b_t, s). & \text{(Bounding)} \end{aligned}$$

[1] Other authors use $0 \leq b_t < n$, but we can often avoid special cases with this definition.
[2] This is essentially [53, Prop. 3.9].

Furthermore, if P is valid, then $f(x)$ has P as its ramification polygon if and only if $F_{b_t} = \ell_P(b_t, s_t)$ for all t and $F_i \geq \ell_P(i, s)$ whenever $p^s \leq i \leq n$.[3]

Note that the three "Ore conditions" are so-called because they imply

$$\min(nB(b_t, p^{s_t}), nB(n, p^{s_t})) \leq J_t \leq nB(n, p^{s_t})$$

which for $t = 0$ is the bound $\min(nv(b_0), nv(n)) \leq J_0 \leq nv(n)$ discovered by Ore [51].

We define P to be **weakly valid**[4] if it satisfies these same conditions but with s restricted to $\{s_t\}$, i.e.

$$\begin{aligned}
p^{s_t} &\leq b_t \\
b_t = n &\implies \ell_P(n, s_t) = 0 && \text{(Ore 1)} \\
&\ell_P(n, s_t) \leq 0 && \text{(Ore 2)} \\
b_t < n &\implies \ell_P(b_t, s_t) \geq 1 && \text{(Ore 3)} \\
b_r = b_t < n &\implies \ell_P(b_t, s_t) = \ell_P(b_r, s_r) && \text{(Consistency)} \\
p^{s_r} \leq b_t < n &\implies \ell_P(b_t, s_t) \geq \ell_P(b_t, s_r). && \text{(Bounding)}
\end{aligned}$$

Noting that $\ell_P(i, s_t) = a_t + \left\lceil \frac{b_t - i}{n} \right\rceil - B(i, p^{s_t}) + 1$ is actually a function of i and the vertex $(p^{s_t}, J_t = a_t n + b_t)$, we deduce that if the vertices of P are a subset of the vertices of $\hat{P}$ and $\hat{P}$ is weakly valid, then P is also weakly valid. In particular if P is valid, then removing vertices gives a weakly valid polygon.

2.4 Enumeration

We can use the Ore bounds to enumerate all possible ramification polygons P for a given degree n. We perform a branching algorithm to assign all possible combinations of vertices. The state of a branch is a paritally-assigned polygon P and an integer $0 \leq S \leq v_p(n)$. First, we branch over each $J_0 = 0, 1, \ldots, nv(n)$ satisfying the Ore bound, and set the state to $P = [(1, J_0), (p^{v_p(n)}, 0), (n, 0)]$ and

[3] This is essentially [53, Prop. 3.10].

[4] This is new.

$S = 1$. Given the state $P = [(p^{s_0} = 1, J_0), \ldots, (p^{s_k}, J_k), (p^{v_p(n)}, 0), (n, 0)]$ and $s_k < S < v_p(n)$ we consider adding a vertex of the form $(p^S, *)$ to P or not. Hence we branch: either we don't add a vertex, in which case the state of the branch becomes P and $S + 1$; otherwise we branch for each possible new vertex (p^S, J) with $0 < J < P(p^S)$, in which case the state becomes $P + (p^s, J)$ and $S + 1$. Finally, if we are given the state P and $S = v_p(n)$ then we have decided on all vertices for P, and so we check it is valid and if so, output it.

This algorithm is quite impractical but can be made practical by terminating a branch if the current P is not weakly valid. We know that removing vertices from a valid polygon preserves weak validity, so if P is not weakly valid it can not be augmented to a valid one.[5]

For example, to compute all 340 ramification polygons of degree 16 over $\mathbb{Q}_2$ would require considering around 300,000 branches, but can reduce this to 1602 branches by terminating using weak validity. The 4948 ramification polygons of degree 32 are found with 29,730 branches instead of around 600,000,000. Our implementation computes these in around 3 and 160 seconds respectively.

2.5 Template

Suppose we write $f_i = \sum_{0 \le k \le \infty} \hat{f}_{i,k} \pi^k$ for $f_{i,k} \in \mathbb{F}_K$, where $\hat{\cdot} : \mathbb{F}_K \to \mathcal{O}_K$ is a choice of representative. A **template** is a collection of sets $X_{i,k} \subset \mathbb{F}_K$ defining a set of monic polynomials[6]

$$X = \left\{ x^n + \sum_{i=0}^{n-1} x^i \sum_{k=0}^{\infty} \hat{f}_{i,k} \pi^k \ : \ f_{i,k} \in X_{i,k} \right\} \subset K[x].$$

For example, the template for all Eisenstein polynomials has $X_{i,0} = \{0\}$, $X_{0,1} = \mathbb{F}_K^\times$, otherwise $X_{i,k} = \mathbb{F}_K$.

[5] This algorithm is similar to one described in Sinclair's thesis [62, §3.3]. It is not clear if that algorithm is correct because it does not seem to use our concept of weak validity, which appears necessary for building up polygons one vertex at a time. Furthermore, tables [64] produced from their implementation are missing some polygons. For example, the tables for degree 8 extensions of $\mathbb{Q}_2$ are missing the polygon $[(1, 7), (2, 6), (4, 4), (8, 0)]$, which is the ramification polygon of $x^8 + 2x^7 + 2x^6 + 2x^4 + 2$.

[6] Enumerating Eisenstein polynomials via templates is used throughout [53], and in particular in Algorithm 6.1.

Given a valid ramification polygon P, a template consisting precisely of the Eisenstein polynomials with ramification polygon P is given by the template for Eisenstein polynomials with additionally $X_{i,k} = \{0\}$ for all $p^s \leq i$, $k < \ell_P(i,s)$, and $X_{b_t,\ell_P(b_t,s_t)} = \mathbb{F}_K^\times$ for all t.

Given a finite template, one can enumerate all of its polynomials by enumerating the cartesian product of the $X_{i,k}$.

One can show as a consequence of Krasner's lemma that if $f, \hat{f} \in K[x]$ are Eisenstein and $f - \hat{f}$ has coefficients of valuation more than $1 + 2J_0/n$ then they generate the same field. Therefore a template can be made finite by setting $X_{i,k} = \{0\}$ for $k > 1 + 2J_0/n$, and its polynomials will between them generate the same set of extensions. In §6 we shall see how to make the template even smaller.

3 Fine ramification polygon

3.1 Definition

We define the set

$$\mathcal{P}^* = \mathcal{P} \cap P = \{(j, R_j) \in \mathcal{P} \,:\, P(j) = R_j\}$$

to be the **fine ramification polygon of** f. It is not strictly a polygon itself, but its lower convex hull is P, and so this is a finer quantity than the ramification polygon because it can include points in the interiors of the faces. As discussed, all points $(j, *)$ for $j \leq p^{v_p(n)}$ in $\mathcal{P}^*$ are of the form $(p^s, *)$ and so we denote $\mathcal{P}^*$ by

$$\mathcal{P}^* = [(p^{s_0} = 1, J_0), \ldots, (p^{s_u}, J_u = 0), \ldots, (n, 0)].$$

Note that the quantities u, s_t and J_t may be different to their earlier meaning in the context of the plain ramification polygon, since there may now be extra points between the vertices. Also note that there are possibly some extra points between $(p^{s_u}, 0)$ and $(n, 0)$.

3.2 Invariant

We shall see in §4.2 that $\mathcal{P}^*$ is an invariant of L/K.

3.3 Validity

A potential fine ramification polygon $\mathcal{P}^*$ is valid if and only if there are $F_i \in \mathbb{Z}$ such that $R_j = J$ for all points $(j, J) \in \mathcal{P}^*$ and for all j without a point above j we have $R_j > P(j)$.

First we consider the horizontal face. For $p^{s_u} \leq j \leq n$ we need that $R_j = 0$ if and only if $(j, 0) \in \mathcal{P}^*$. Observe that $r_j \equiv \binom{n}{j}$ and therefore $R_j = 0$ if and only if $B(n, j) = 0$. Hence the condition for this range of j is that $(j, 0) \in \mathcal{P}^*$ if and only if $B(n, j) = 0$.

For the remaining points $(p^s, *)$, the analysis is almost identical to the case with the plain ramification polygon, the only difference being that we require $R_{p^s} > P(p^s)$ whenever there is not a point $(p^s, *) \in \mathcal{P}^*$. Following the same steps, we deduce the following conditions:

$$\begin{aligned} & & F_0 &= 1 \\ 0 < i < n & \implies & F_i &\geq 1 \\ & & F_n &= 0 \\ p^s \leq i \leq n & \implies & F_i &\geq \ell_{\mathcal{P}^*}(i, s) \\ & & F_{b_t} &= \ell_{\mathcal{P}^*}(b_t, s_t) \end{aligned}$$

where

$$\ell_{\mathcal{P}^*}(i, s_t) := a_t - B(i, p^{s_t}) + 1 + 1[i < b_t]$$

and for $(p^s, *) \notin \mathcal{P}^*$

$$\ell_{\mathcal{P}^*}(i, s) := \left\lfloor \frac{P(p^s) - i}{n} \right\rfloor - B(i, p^s) + 2.$$

Observing that these conditions are essentially identical to those from the previous section, up to the redefinition of $\ell_{\mathcal{P}^*}$, then we similarly find that $\mathcal{P}^*$

is valid if and only if[7]

$$\begin{aligned}
B(n,j) = 0 &\iff (j,0) \in \mathcal{P}^* && \text{(Tame)}\\
&\qquad p^{s_t} \le b_t\\
b_t = n &\implies \ell_{\mathcal{P}^*}(n, s_t) = 0 && \text{(Ore 1)}\\
&\qquad \ell_{\mathcal{P}^*}(n, s) \le 0 && \text{(Ore 2)}\\
b_t < n &\implies \ell_{\mathcal{P}^*}(b_t, s_t) \ge 1 && \text{(Ore 3)}\\
b_r = b_t < n &\implies \ell_{\mathcal{P}^*}(b_t, s_t) = \ell_{\mathcal{P}^*}(b_r, s_r) && \text{(Consistency)}\\
p^s \le b_t < n &\implies \ell_{\mathcal{P}^*}(b_t, s_t) \ge \ell_{\mathcal{P}^*}(b_t, s). && \text{(Bounding)}
\end{aligned}$$

If $\mathcal{P}^*$ is valid, then $f(x)$ has $\mathcal{P}^*$ as its fine ramification polynomial if and only if $F_{b_t} = \ell_{\mathcal{P}^*}(b_t, s_t)$ for all t and $F_i \ge \ell_{\mathcal{P}^*}(i, s)$ whenever $p^s \le i \le n$.[8]

We define $\mathcal{P}^*$ to be **weakly valid** if these conditions hold only for $s \in \{s_t\}$. As before, weak validity is preserved under removing points from $\mathcal{P}^*$.

Enumerating the possible fine ramification polygons and producing their templates is essentially the same as for plain ramification polygons.

4 Residues

4.1 Definition

We define ρ_j to be the leading α-adic coefficient of r_j. Recall that if $R_j := nv(r_j) = an + b$ then $r_j \sim \binom{b}{j} f_b \alpha^{b-n}$. Hence

$$\rho_j \equiv r_j \alpha^{-R_j} \equiv \binom{b}{j} (\phi_b \pi^{F_b})(-\phi_0 \pi)^{-1-a}$$

where $\phi_i \equiv f_i \pi^{-F_i}$.

We refer to $-\phi_0$ as the **uniformizer residue of α (with respect to π)**

[7] This is essentially [53, Prop. 3.9], which is slightly mis-stated because it does not include the Ore 2 condition in the case that there is no point $(p^s, *) \in \mathcal{P}^*$.

[8] This is essentially [53, Prop. 3.10].

because $\alpha^n \sim -f_0 \sim (-\phi_0)\pi$.

We extend the points of the fine ramification polygon to include the residues (j, R_j, ρ_j), giving the **fine ramification polygon with residues**[9]

$$\mathcal{P}^*_{\text{res}} = [(p^{s_0} = 1, J_0, \gamma_0), \ldots, (p^{s_u}, 0, \gamma_u), \ldots, (n, 0, 1)].$$

4.2 Invariant

If the ramification polygon P has a face $((j_0, R_{j_0}), (j_1, R_{j_1}))$ of width $w = j_1 - j_0$ and slope $(R_{j_1} - R_{j_0})/w = -h/e$, we define its **residual polynomial** to be

$$A(x) \equiv r_{j_1}^{-1} \sum_{i=0}^{w/e} r_{j_0+ie} \alpha^{jh - R_{j_0}} x^i \equiv \sum_{\substack{(j,R_j,\rho_j)\in\mathcal{P}^*_{\text{res}} \\ j_0 \le j \le j_1}} \rho_j x^{(j-j_0)/e} \in \mathbb{F}_K[x].$$

It is well-known that the w roots $\frac{\alpha - \sigma\alpha}{\alpha}$ of r of valuation h/e satisfy

$$A((\tfrac{\alpha-\sigma\alpha}{\alpha})^e/\alpha^h) \equiv 0.$$

Observe that A, being monic, is determined by its roots. Recalling that if we choose another uniformizer of $\hat{\alpha} \in L = K(\alpha)$ then $\frac{\hat{\alpha}-\sigma\hat{\alpha}}{\hat{\alpha}} \sim \frac{\alpha-\sigma\alpha}{\alpha}$, these are the roots of $r(x)$ and $\hat{r}(x)$, and we deduce that if $\hat{A}$ is the corresponding residual polynomial then $\hat{A}(x) \equiv A(\delta^h x)$ where $\hat{\alpha} \sim \delta\alpha$.

Firstly this implies that a coefficient of $\hat{A}$ is zero if and only if the corresponding coefficient of A is zero. This implies that $\mathcal{P}^* = \hat{\mathcal{P}}^*$ is an invariant of L/K.

Furthermore, this implies that $\hat{\rho}_j \equiv \rho_j \delta^{-R_j}$ for $(j, R_j, \rho_j) \in \mathcal{P}^*_{\text{res}}$. Therefore, given $\mathcal{P}^*_{\text{res}}$ and $\hat{\mathcal{P}}^*_{\text{res}}$, we consider them equivalent if they are equal as fine ramification polygons and if there exists $\delta \in \mathbb{F}_K^\times$ such that $\hat{\rho}_j \equiv \rho_j \delta^{-R_j}$. Then equivalence classes are an invariant of L/K.[10]

Note that $\hat{\rho}_j \equiv \rho_j \delta^{-R_j}$ if and only if $\hat{r}_j \sim r_j$ when $\hat{\alpha} \sim \delta\alpha$. Therefore being equivalent is the same thing as $v(r_j - \hat{r}_j) > P(j)$ for all $1 \le j \le n$.

[9] In [53], the equivalent notion of residual polynomials is studied instead, but this additional structure is not actually used.

[10] This is equivalent to the invariant $\mathcal{A}$ of [53, §4.1].

4.3 Validity

Given a fine ramification polygon with residues $\mathcal{P}^*_{\text{res}}$, then it is valid if and only if: (a) it is valid as a fine ramification polygon; (b) for $(j, 0, \rho_j) \in \mathcal{P}^*_{\text{res}}$ we have $\rho_j \equiv r_j \equiv \binom{n}{j}$; and (c) there are $\phi_i \in \mathbb{F}_K^\times$ such that for each t

$$\gamma_t \equiv \binom{b_t}{p^{s_t}} (\phi_{b_t} \pi^{F_{b_t}})(-\phi_0 \pi)^{-1-a_t} \equiv \beta(b_t, p^{s_t}) \phi_{b_t} (-\phi_0)^{-1-a_t}$$

where $\beta(i, j) \equiv \binom{i}{j} \pi^{-B(i,j)}$.

Eliminating ϕ_i from these latter equations for $0 < i \leq n$, we find they are soluble if and only if there is $\phi_0 \in \mathbb{F}_K^\times$ such that[11]

$$\begin{aligned} b_t = n &\implies \gamma_t \equiv \beta(n, p^{s_t})(-\phi_0)^{-a_t-1} \\ b_t = b_r < n &\implies \frac{\gamma_t}{\gamma_r} \equiv \frac{\beta(b_t, p^{s_t})}{\beta(b_r, p^{s_r})} (-\phi_0)^{a_r - a_t}. \end{aligned}$$

If $\mathcal{P}^*_{\text{res}}$ is valid, then $f(x)$ has $\mathcal{P}^*_{\text{res}}$ as its fine ramification polygon with residues if and only if ϕ_0 is a solution to these equations and

$$\phi_{b_t} \sim \gamma_t \beta(b_t, p^{s_t})^{-1} (-\phi_0)^{a_t+1}$$

for each t.

4.4 Enumeration

Given $\mathcal{P}^*$, we can enumerate all possible $\mathcal{P}^*_{\text{res}}$ extending it by initially having γ_t unset for all t. Similar to our branching algorithm for enumerating ramification polygons, we branch for each t over the possible values of γ_t.

We can make this practical by terminating a branch if the current partial assignment of γ_t is not **weakly valid**, where a partial assignment is weakly valid if there exists $\phi_0 \in \mathbb{F}_K^\times$ such that the above conditions involving only the assigned γ_t are true.

[11] This is essentially [53, Prop. 4.5], which is slightly mis-stated because it misses the conditions in the case $b_t = n$.

Suppose we have assigned $\gamma_0, \ldots, \gamma_{t-1}$ so far and are branching over possibilities for γ_t. We can restrict the algorithm to produce one representative $\mathcal{P}^*_{\text{res}}$ per class by considering γ_t equivalent to $\hat{\gamma}_t$ if there exists $\delta \in \mathbb{F}_K^\times$ such that $\delta^{-J_k} \equiv 1$ for $k < t$ and $\hat{\gamma}_t \equiv \gamma_t \delta^{-J_t}$, and only branching over representatives of equivalence classes.

4.5 Template

The choice of ϕ_0 means there is not a template (by our definition) for Eisenstein polynomials corresponding to $\mathcal{P}^*_{\text{res}}$. This will be fixed in the next section by including ϕ_0 in the invariant.

5 Uniformizer residue

5.1 Invariant

We now consider the extent to which the pair $(\mathcal{P}^*_{\text{res}}, \phi_0)$ is an invariant of L/K. We have seen that changing uniformizer $\alpha \to \hat{\alpha} \sim \delta\alpha$ changes $\rho_j \to \hat{\rho}_j \sim \delta^{-R_j}\rho_j$. Since by definition $\alpha^n \sim -\phi_0\pi$, we find that $\phi_0 \to \hat{\phi}_0 \sim \delta^n\phi_0$.

Therefore we say $(\mathcal{P}^*_{\text{res}}, \phi_0)$ and $(\hat{\mathcal{P}}^*_{\text{res}}, \hat{\phi}_0)$ are equivalent if there is $\delta \in \mathbb{F}_K^\times$ such that $\hat{\rho}_j \sim \delta^{-R_j}\rho_j$ for all $(j, R_j, \rho_j) \in \mathcal{P}^*_{\text{res}}$ and $\hat{\phi}_0 \sim \delta^n\phi_0$. Equivalence classes are an invariant for L/K.[12]

By the preceding section, ϕ_{b_t} is determined for all t by the choice of representative. Therefore considering $(\mathcal{P}^*_{\text{res}}, \phi_0, \{\phi_{b_t} : t\})$ does not give a finer invariant.

5.2 Validity and enumeration

Suppose we are given a valid $\mathcal{P}^*_{\text{res}}$, then the valid ϕ_0 which extend this are precisely the solutions to the system of the previous section. Hence to enumerate them all, we find the solutions.

If we want to find representatives of equivalence classes, note that $(\mathcal{P}^*_{\text{res}}, \phi_0)$ and $(\mathcal{P}^*_{\text{res}}, \hat{\phi}_0)$ are equivalent if there is δ such that $\delta^{-g} = 1$ and $\delta^n = \hat{\phi}_0/\phi_0$ where

[12] This is equivalent to the invariant $\mathcal{A}^*$ of [53, §4.2].

$g = \gcd_t J_t = \gcd_{(j,R_j)\in\mathcal{P}^*} R_j$. We use this criterion to test for equivalence, and so choose one ϕ_0 per equivalence class.

5.3 Template

The template for Eisenstein polynomials corresponding to $(\mathcal{P}^*_{\text{res}}, \phi_0)$ is the template for $\mathcal{P}^*_{\text{res}}$ with the changes $X_{0,1} = \{\phi_0\}$ and

$$X_{b_t,\ell_{\mathcal{P}^*}(b_t,s_t)} = \{\gamma_t \beta(b_t, p^{s_t})^{-1}(-\phi_0)^{a_t+1}\}.$$

6 Change of uniformizer

So far we have studied the change of uniformizer $\alpha \to \hat{\alpha} \sim \delta\hat{\alpha}$ and the effect of $\bar{\delta}$. We now consider smaller perturbations, i.e. $\hat{\alpha} = \alpha(1 + u\alpha^m)$ for $m > 0$, $u \in \mathcal{O}_L$. These will allow us to considerably reduce the size of our templates.

Letting $\hat{f}(x)$ be the minimal polynomial for $\hat{\alpha}$ then

$$f(\hat{\alpha}) - \hat{f}(\hat{\alpha}) = f(\hat{\alpha}) = \alpha^n r(\tfrac{\hat{\alpha}}{\alpha} - 1) = \alpha^n r(u\alpha^m)$$

and

$$f(\hat{\alpha}) - \hat{f}(\hat{\alpha}) = \sum_{i=0}^{n-1} (f_i - \hat{f}_i)\hat{\alpha}^i.$$

Writing

$$S_m(x) \equiv \alpha^{-C_m} r(\alpha^m x) \in \mathbb{F}_K[x]$$

with C_m as large as possible, and writing $C_m = c_m n + d_m$ with $0 \le d_m < n$ then

$$S_m(u) \equiv (f_{d_m} - \hat{f}_{d_m})\alpha^{d_m - n - C_m} \equiv (f_{d_m} - \hat{f}_{d_m})(-\phi_0\pi)^{-1-c_m}.$$

For $m > 0$, S_m only consists of terms from points on faces of $\mathcal{P}^*$ with negative slope, i.e. the points (p^{s_t}, J_t), and hence S_m only has terms at powers of p and so is additive. Therefore it defines a $\mathbb{F}_p$-linear map $S_m : \mathbb{F}_K \to \mathbb{F}_K$.

By changing uniformizer $\alpha \to \hat{\alpha}$ we change $f_{d_m,1+c_m}$ by $(-\phi_0)^{1+c_m} S_m(u)$. Therefore if we restrict the template by setting $X_{d_m,1+c_m}$ to be a set of coset representatives of $\mathbb{F}_K^+/(-\phi_0)^{1+c_m} S_m(\mathbb{F}_K^+)$, then it will produce the same set of fields.

Observe that since C_m is strictly increasing in m (in fact C_m is essentially the Hasse-Herbrand transition function; see e.g. [60, 35]) then this change to the template is independent to the changes made for any higher m, and so we can make all of these changes independently.

In particular, if m is higher than the negative of the slope of the steepest face of P then $S_m(x) = x$ is surjective, and so we may set $X_{d_m,1+c_m} = \{0\}$. In this region $C_m = J_0 + m$, and so this sets all but finitely many $X_{i,k} = \{0\}$. Note that this change is independent of ϕ_0 (unlike for smaller m) and therefore can be used when enumerating polynomials from simpler invariants like P or $\mathcal{P}^*$.

7 Implementation notes

7.1 Representation of invariants

Our representation of a ramification polygon is a little more general than is presented in this paper. We actually represent a polygon as a list

$$[(x_0, J_0, \sim_0, \rho_{x_0}), (x_1, J_1, \sim_1, \rho_{x_1}), \ldots, (n, 0, *, \rho_n)]$$

where $x_i = p^i$ for $i \leq v_p(n)$. That is, we have a point for every power of p. The relation $\sim_i$ specifies that $R_{x_i} \sim_i J_i$ where $\sim_i$ is $=$, $\geq$ or $>$. Hence if $\sim_i$ is $=$ then (x_i, J_i) is a point of the (fine) ramification polygon; if it is $>$ then it is not; if it is $\geq$ then it is unspecified.

In particular, for the plain ramification polygon, $\sim_i$ is $=$ at the vertices and $\geq$ elsewehere. For the fine ramification polygon, $\sim_i$ is $=$ or $>$.

The residues ρ_{x_i} may be unspecified. Where they are specified, they must be non-zero and $\sim_i$ must be $=$.

The arguments from earlier sections are simply modified to yield validity and equivalence conditions for this more general object. One way in which this is useful is that we can now specify one residue at a time, and check for validity at each step, and therefore the branching algorithms for enumerating these invariants may be implemented.

7.2 Consistency of roots

Checking for equivalence of ramification polygons with residues requires solving a system of equations of the form $x^{k_i} = a_i$.

We compute the extended GCD

$$K := \gcd(k_1, k_2, \ldots) = \sum_i b_i k_i$$

so that

$$x^K = \prod_i x^{b_i k_i} = \prod_i a_i^{b_i} =: A$$

and

$$a_i = x^{k_i} = x^{K(k_i/K)} = A^{k_i/K}.$$

We deduce the system is solvable if and only if $a_i = A^{k_i/K}$ and $x^K = A$ is solvable. Hence we can check for consistency and reduce the system to a single equation.

To limit the solutions to $\mathbb{F}_K^\times$, we let $q = |\mathbb{F}_K|$ and include $x^{q-1} = 1$ in the system of equations.

7.3 Binomial coefficients

A little care is needed to compute with binomial coefficients π-adically.

One can see that

$$v_p(k!) = \sum_{i=1}^{\infty} \left\lfloor \tfrac{k}{p^i} \right\rfloor$$

and from which we can compute

$$B(r,k) = e(K/\mathbb{Q}_p)(v_p(r!) - v_p(k!) - v_p((r-k)!)).$$

By Wilson's theorem, if $k = ap + b$ then the "p-unit" part of the factorial is

$$U_p(k) := \prod_{1 \le i \le k, v_p(i)=0} i \equiv (-1)^a b! \mod p$$

from which we can compute the "p-shifted factorial"

$$S_p(k) := k!/p^{v_p(k!)} \equiv \prod_{i=1}^{\infty} U_p(\left\lfloor \tfrac{k}{p^i} \right\rfloor) \mod p$$

and hence

$$\beta(r,k) \equiv \frac{S_p(r)}{S_p(k)S_p(r-k)}\gamma^{-v_p\binom{r}{k}}$$

where $\pi^{e(K/\mathbb{Q}_p)} \sim \gamma p$ (i.e. γ is the uniformizer residue of π with respect to p c.f. §4.1).

Chapter IV

Exact p-adics

Foreword

This chapter has been published separately as an article [22].

1 Introduction

When dealing with completed fields, such as $\mathbb{R}$ or $\mathbb{Q}_p$, it is generally quite difficult to represent elements exactly. Instead, the commonest way to represent elements is by specifying them to some pre-determined precision, and then performing operations such as arithmetic to this precision also. This is the foundation of floating point arithmetic. For example, one might represent the real number e by its approximation 2.718281828 to a precision of 10 real digits. We say such a representation is **inexact** because several real numbers can have the same representation: e, 2.718281828 and 2.7182818281 all have the same representation to 10 digits precision.

Such a representation is also usually **zealous** meaning that when an operation is performed, such as multiplication, it is immediately computed to the required precision. For instance, computing $e \times e$ will work to 10 digits precision and actually compute $2.718281828 \times 2.718281828 = 7.389056096$. In fact, $e \times e = 7.389056098\ldots$, demonstrating that precision errors can creep into the results, so that they are in fact less precise than the precision claims.

An often-suggested alternative to zealous arithmetic is **lazy arithmetic**, in which an operation does not produce an answer per-se, but a "promise to produce an answer to a desired precision". That is, calling $e \times e$ would not produce the approximation 2.718281828, but would produce a function which, when called with an integer k, returns an approximation to $e \times e$ to k digits precision.

Such a function can be said to be an **exact** representation of a real number, because no two distinct real numbers have the same representation: for a sufficiently large precision k, the representing functions will return different approximations.

These comments hold true for p-adic numbers too. For instance, an element of $\mathbb{Q}_p$ is generally represented in zealous, inexact arithmetic by its residue class in $\mathbb{Q}_p/p^k\mathbb{Z}_p$ for some absolute precision k: e.g. $1 + 2^{10}\mathbb{Z}_2$ might represent 1, $1 + 2^{10}$ or $1 + 5 \times 2^{100}$.

There are numerous implementations of such p-adic arithmetic. FLINT [34] provides some low-level arithmetic with elements of $\mathbb{Q}_p$, univariate polynomials over $\mathbb{Q}_p$, and unramified extensions of $\mathbb{Q}_p$. Sage [57] and Magma [8] have more fully-featured implementations, including arbitrary finite extensions of $\mathbb{Q}_p$ and higher-level routines for tasks such as factoring.

Also of note is an implementation in Mathemagix [38] of the so-called **relaxed p-adic arithmetic**, which treats elements of $\mathbb{Q}_p$ like an infinite sequence of p-adic coefficients, somewhat like $\mathbb{F}_p((t))$, and represents them by a truncated sequence followed by a function to retrieve the next coefficient. This representation is therefore exact, because for different numbers, these streams of digits must eventually diverge. This has specific uses in p-adic recursion solving, and in principle is useful in general, but is somewhat more complicated to implement than the lazy arithmetic presented in this article, and as such is less fully featured.

A more in-depth description of different p-adic arithmetic systems is given by Caruso [10].

In this article, we present two new implementations of two different lazy, exact p-adic arithmetic systems. The implementations are written for the Magma computer algebra system [8] which, as mentioned above, already has a fully-featured implementation of zealous, inexact p-adic arithmetic. Our packages, called `ExactpAdics` and `ExactpAdics2`, aim to use the inexact functionality

already available as much as possible, in order to provide a more user-friendly wrapper. This allows for rapid addition of new features to the exact arithmetic as soon as they are available inexactly.

To the author's knowledge, these are the first highly-featured, general-purpose implementations of lazy p-adic arithmetic.

This article describes the rationale and the fundamental concepts behind the packages, but does not constitute a user manual. The user manuals are available online at [21] and [23] and the packages may be downloaded from here also.

At the time of writing, we recommend the typical user to use the `ExactpAdics2` package (§6).

As an application, these packages have been used to implement the algorithm in Chapter V to compute the 2-part of the conductor of a hyperelliptic curve of genus 2 defined over a number field. This implementation is available from [24]. It uses such high-level p-adic routines as: computing the completion of a number field at a finite place (§9.4); computing the factorization of a univariate polynomial (§9.12) and the fields defined by its factors; and Hensel-lifting roots of a system of multivariate equations (§9.9).

As another application, these packages can optionally be used with the implementation of the algorithms described in Chapter II for computing the Galois group of a p-adic polynomial. This is available from [27]. With either package present the Galois group algorithm becomes provably correct, whereas otherwise with inexact p-adics there is no such guarantee. We also find that the algorithms run faster with exact p-adics, at least for reasonably high-degree inputs.

1.1 Terminology

Suppose K is a p-adic field (a finite extension of $\mathbb{Q}_p$), with ring of integers $\mathcal{O} = \mathcal{O}_K$ and uniformizing element $\pi = \pi_K$. The π-adic valuation is denoted $\mathrm{val} = \mathrm{val}_K$ such that $\mathrm{val}(\pi) = 1$.

When we refer to an **inexact** (representation of a) p-adic number $x \in K$, we mean a conjugacy class $x + \pi^k\mathcal{O}$. We refer to k as the **absolute precision** of (the representation of) the number.

Equivalently, it may be represented as $\pi^v(y + \pi^r\mathcal{O})$ where $y \in \mathcal{O}$ and $r \geq 0$. We

refer to v as the **weak valuation** of x; it is a lower bound on the true valuation of x. We refer to r as the **relative precision**; it bounds the number of non-zero π-adic digits of x known. Note that $v + r = k$.

We say that x is **weakly zero** if $y \in \pi^r\mathcal{O}$, that is if the representation is of the form $\pi^{v+r}\mathcal{O}$. Note:

- If x is not weakly zero, then it is not zero.
- If $r = 0$ then x is weakly zero.

We typically enforce the following normalizing condition: if $r > 0$ then $y \in \mathcal{O}^\times$. Now note:

- If x is not weakly zero, then its valuation is exactly v.
- x is weakly zero if and only if $r = 0$ (and if and only if $k = v$).

Magma's builtin p-adics (`FldPad`, `FldPadExact`, etc.) are inexact in this sense, and satisfy the normalizing condition. We note that *prime* p-adic fields — i.e. $\mathbb{Q}_p$ — as opposed to their elements, can themselves naturally be represented exactly by the prime itself. Extensions of the form $K(x)/(f(x))$ are usually represented inexactly via an inexact representation of the polynomial $f(x) \in K[x]$; however we note that Magma does additionally have a builtin exact representation of extensions, represented by a map $m : \mathbb{Z} \to K[x]$ such that $m(k)$ is a defining polynomial to precision k. We refer to this latter representation as **semi-exact**, since the field is represented exactly but its elements are represented inexactly.

The residue class field $\mathcal{O}/\pi\mathcal{O}$ is denoted $\mathbb{F} = \mathbb{F}_K$, and $\bar{x} \in \mathbb{F}$ denotes the residue class of $x \in \mathcal{O}$.

A polynomial $f(x) = \sum_{i=0}^{d} f_i x^i \in K[x]$ of degree d is **Eisenstein** if $\mathrm{val}(f_0) = 1$, $\mathrm{val}(f_i) \geq 1$ for $1 \leq i < d$ and $\mathrm{val}(f_d) = 0$. It is irreducible, its roots have valuation $\frac{1}{d}$, and so it defines a totally ramified extension $K(x)/(f(x))$ of degree d such that $x + (f(x))$ is a uniformizer.

A polynomial $f(x) = \sum_{i=0}^{d} f_i x^i \in \mathcal{O}[x]$ of degree d is **inertial** if $\mathrm{val}(f_d) = \mathrm{val}(f_0) = 0$ and $\bar{f}(x) = \sum_{i=0}^{d} \bar{f}_i x^i \in \mathbb{F}[x]$ is irreducible over the residue class field $\mathbb{F}$. It is irreducible, the residue classes of its roots generate an extension of $\mathbb{F}$ of degree d, and so it defines an unramified extension $K(x)/(f(x))$.

1.2 Comparison of zealous and lazy arithmetic

Precision

In zealous arithmetic, the user is generally required to choose a precision to work at in advance. Then all computations are performed to that precision, and it may happen that the precision chosen was not sufficient. In this case, the user will probably start the computation over with a higher precision. This process of manually increasing the precision of a computation can be burdensome for the user. In lazy arithmetic, such precision decisions are made automatically as far as possible.

Example 1.1. Here is a typical interactive Magma session, using its builtin lazy arithmetic:

```
> // try to factorize at precision 10
> K := pAdicField(2, 10);
> R<x> := PolynomialRing(K);
> f := my_favourite_polynomial(R);
> Factorization(f);
error: ...
> // try to factorize at precision 20
> K := pAdicField(2, 20);
> R<x> := PolynomialRing(K);
> f := my_favourite_polynomial(R);
> Factorization(f);
error: ...
> // try to factorize at precision 40
> K := pAdicField(2, 40);
> R<x> := PolynomialRing(K);
> f := my_favourite_polynomial(R);
> Factorization(f);
[ <x^10 + ... >, ... ]
```

Using lazy arithmetic provided by our package, the equivalent session would be the following. Note that there is no explicit mention of precision.

```
> K := ExactpAdicField(2);
> R<x> := PolynomialRing(K);
> f := my_favourite_polynomial(R);
> Factorization(f);
[ <x^10 + ... >, ... ]
```
$\Diamond$

In lazy arithmetic, each individual computation is performed to approximately the smallest precision it can be, and so precisions are very "local" in the computation. In zealous arithmetic, the precision is generally chosen once at the start of a computation, so each operation is performed to the same precision, and so precisions are more "global". If there is a single operation requiring a high "global" precision, this increases the precision that all other operations are performed to, which is a performance hit compared to lazy arithmetic.

Example 1.2. An example comes from the conductor algorithm mentioned in the introduction. One portion of this algorithm takes a polynomial $f(x) \in \mathbb{Q}_2[x]$, computes its factorization, chooses a factor $g(x)$, computes the extension $L/\mathbb{Q}_2$ defined by g, and then finds a root of g in L. Usually, the precision required for the factorization far exceeds that of the root-finding; however, because the root-finding is over an extension L, if it were to be done at the same high precision as the factorization, its run-time would often dominate. $\Diamond$

Correctness and provability

When a p-adic number $x \in K$ is represented inexactly as a class $x + \pi^k\mathcal{O}$, then it can be ambiguous whether it is really representing x or the class itself. For many operations, the distinction makes no difference; for example since

$$(x + y) + \pi^k\mathcal{O} = (x + \pi^k\mathcal{O}) + (y + \pi^k\mathcal{O})$$

then addition works the same in either interpretation. For other operations, Magma can produce potentially misleading answers; for example if x is represented as $0 + \pi^k\mathcal{O}$ then `Valuation(x)` will return k, when in fact all we really know is that $\mathrm{val}(x) \geq k$.

Definition 1.3. Suppose F is a mathematical function and suppose $\tilde{F}$ is a programmatic function intended to implement F, so it takes as inputs representations of the inputs of F and returns as outputs representations of the outputs of F. We say that $\tilde{F}$ **represents** F if for all possible inputs X to F and representations $\tilde{X}$ of X, that $\tilde{F}(\tilde{X})$ either does not return successfully or returns a representation of $F(X)$.

Hence if $\tilde{F}$ represents F, then its outputs depend only on the inputs being represented, and not on the representation of the inputs themselves. In the case of p-adic computation, this means that the output of $\tilde{F}$ should not depend on the precision that its inputs were given to, and therefore is unambiguously a function of the p-adic value, and not its representation.

As already indicated, the `Valuation` intrinsic in Magma does not represent the valuation function. Also equality is not represented, because it actually is equality of the representation: if $x = 1$ and $y = 1 + 2^{10}$ are both represented by $1 + 2^{10}\mathbb{Z}_2$ then `x eq y` will be true. In fact, it is not possible to determine that two p-adic numbers are equal when given to any finite precision, and it is only possible to tell that they are unequal if they are given to sufficiently large precision.

As another example, given a polynomial $f(x)$ represented as $\tilde{f}(x) = (1 + \pi^{10}\mathcal{O})x^2 + (0 + \pi^{10}\mathcal{O})$, the `Roots` intrinsic in Magma will return a double root $\tilde{r} = 0 + \pi^{10}\mathcal{O}$ in K. This is correct as a function of the representations themselves, since $\tilde{f}(\tilde{r}) = 0 + \pi^{10}\mathcal{O}$ represents 0, but if $f(x) = x + 2^{11}$ then it is irreducible and therefore has no roots in K. Similarly `Factorization` and `GCD` do not represent factorization and greatest common divisor of p-adic polynomials.

In our packages, if the name of an intrinsic function is the name of a mathematical function, then the intrinsic represents the function. For example, our `Valuation` intrinsic (see Examples 4.1 and 5.8) will only return the true valuation of the given number; if the input is weakly zero, it may try to increase its precision, and could potentially do this forever (if the input is 0) or raise a precision error, but it is guaranteed that if it returns, its return value is correct.

In some cases, such as `Roots` (§9.8) and `Factorization` (§9.12), the correctness of the output is forced by the fact that the outputs are given exactly. That is, if `Roots` returns a root (exactly), then it by definition comes with a program to

compute an approximation to the root to arbitrarily high precision, and therefore assuming the program is correct this is a proof that the root is correct. In the case of `Roots`, it is Hensel's lemma which provides this proof.

The intrinsics which do not represent a function, and therefore depend on the representation, are given names which make this clear. The terms `Weakly` and `Definitely` are used to denote tests which can give false positives or false negatives; for example `IsWeaklyZero` is true if its input appears to be zero up to some precision (but does not guarantee it is zero), and `IsDefinitelyPrimitive` returns true if its input can be proven to be a primitive element (but if it returns false, this does not imply that its input is not primitive). Similarly the term `Weak` denotes non-representing functions, so `WeakValuation` returns the k in $0 + \pi^k \mathcal{O}_K$ and is therefore actually a lower bound on the true valuation; and `WeakDegree` returns an upper bound on the degree of a polynomial, but which may be incorrect if its top coefficient is actually zero.

Overheads

The main down-sides of lazy arithmetic are the extra time and memory overheads introduced. In lazy arithmetic, p-adic values depend on other p-adic values, and all these dependencies need to be kept in memory for the duration of a computation. Each time an operation is performed, some dependency tracking and propagation needs to occur, which entails some processing time overhead.

This said, we find that these overheads do not usually dominate the run-time of lazy p-adic arithmetic unless one performs a large number of ordinarily very fast operations, such as basic arithmetic. If this is the case, then one can consider implementing the whole sequence of operations as a new atomic p-adic operation, which therefore now only contributes a single node to the dependency graph.

1.3 Structure of this article

The first three sections describe the `ExactpAdics` package.

In §2 we describe the core data types and functionality provided by the package, including a simplified description of the lazy evaluation of p-adic numbers.

In §3 we describe the lazy evaluation scheme actually employed by the package, which includes tracking dependencies between different p-adic values.

In §4 we describe "precision strategies", which are a way of programatically avoiding precision errors with minimal input from the user. This is not a core feature, but greatly improves user-friendliness.

Next we describe the `ExactpAdics2` package and compare.

In §5 we describe the core data types and functionality provided by the package, including a description of the lazy evaluation scheme.

In §6 we compare the merits of the approaches taken by the two packages, including timings on some problems of interest.

The remaining sections describe additional features which either improve user-friendliness or provide more functionality. These features are mainly present in both packages.

In §7 we describe additional structures which are not core functionality, namely multivariate polynomials and tuples.

In §8 we describe our representation of valuations (defined in a generic sense) of p-adic objects, and the operations available on them.

Finally in §9 we give an overview of additional features not covered elsewhere. This is largely to demonstrate that these packages are of practical use, since they include features such as root finding, factorization, residue classes and completions. We also provide some implementation notes.

1.4 Pseudocode

As the package is written in Magma, we shall use a simplification of the Magma language to demonstrate concepts[1]. As this article may be useful to implement similar functionality in other languages, we summarise the syntax here.

Every variable has a **type**. For example a ring of integers has type `RngInt`, an integer has type `RngIntElt`, a boolean (true or false) has type `BoolElt` and an inexact p-adic field has type `FldPad`. New types are defined as

[1]Specifically, we omit `;` and `end`, and imply code blocks through indentation. We also omit `{documentation}` blocks from intrinsics and `declare`.

```
type NAME[ELT]: PARENT
```

where `NAME` is the name of the type. The part in brackets is optional, but when given the type `ELT` is also declared, and `NAME` is actually a structure with elements of type `ELT`. The part after the colon is optional, but when given the new type is a child of `PARENT` in the type hierarchy, and in particular the new type inherits the attributes from `PARENT`.

A type has attributes, which are named pieces of data attached to instances of the type. Attributes may be attached to a type by

```
attributes TYPE: ATTR1, ATTR2, ...
```

where `TYPE` is the name of the type, and `ATTRn` are the names of the attributes.

Instances of the type are created like `x := New(TYPE)`, and its attributes are accessed like `x`ATTR`.

There are three types of functions in Magma: `function`, `procedure` and `intrinsic`, all declared in a similar fashion, such as:

```
function example(x, y : z := 0)
  return x + y + z
```

The difference between the three is that a function returns a value, and should not have any side-effects, a procedure does not return a value but can have side-effects (in particular an input may be passed by reference like `~x` and it becomes modifiable), and an intrinsic is a function or procedure which forms the main user-interface. The `z:=0` part is an optional parameter named `z` whose default value is `0`. Furthermore, intrinsics may have type declarations on its inputs and outputs, which allows overloading of intrinsics with the same name but different type signatures. For example:

```
intrinsic ImportantExpression(
    x :: RngIntElt,
    y :: RngIntElt,
    z :: RngIntElt)
      -> FldRatElt
  return (x^2 + y^2) / z^3
```

is an intrinsic taking three integers and returning a rational.

Note that any pseudocode in this article is illustrative, and does not necessarily match the code in the implementation. The pseudocode is presented as simply as is possible to get the ideas across, whereas the real code will contain more checks and optimizations.

2 ExactpAdics: Core structures and elements

2.1 Abstract base types

In ExactpAdics we have an abstract base type StrPadExact representing any kind of exact p-adic structure or set, and such a set has elements of type PadExactElt:

```
type StrPadExact[PadExactElt]
```

We shall later have sub-types representing the field of p-adic numbers (§2.2), rings of polynomials over p-adic numbers (§2.3), and more (§7).

Such a structure will always have an approximation which is an analogous inexact structure:

```
attributes StrPadExact: approximation
```

Elements always have a parent structure to which they belong, as well as an approximation and an update function:

```
attributes PadExactElt: parent, approximation, update
```

The approximation is an element of the approximation of the parent, and so provides a finite-precision approximation to the element. The update function provides the means to update the approximation arbitrarily precisely, and will be described later in this section and in §3. Figure 1 illustrates the relationships between these attributes.

We provide some universal intrinsics to retrieve the parent of an element, its absolute precision and its weak valuation. The latter two are defined to be the absolute precision and weak valuation of the approximation, and therefore can change over time.

```
intrinsic Parent(x :: PadExactElt) -> StrPadExact
  return x`parent
```

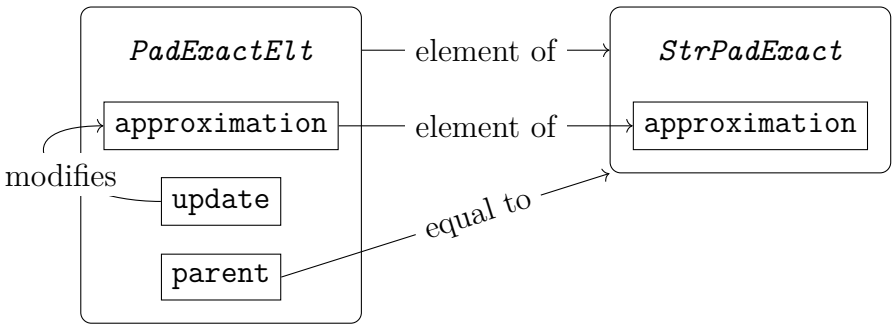

Figure 1: Illustration of the types `StrPadExact` and `PadExactElt`, their attributes and the relationships between them.

```
intrinsic AbsolutePrecision(x :: PadExactElt) -> .
  return AbsolutePrecision(x`approximation)

intrinsic WeakValuation(x :: PadExactElt) -> .
  return WeakValuation(x`approximation)
```

First we describe the representations of p-adic fields and rings of univariate polynomials.

2.2 p-adic fields

An exact p-adic field is represented by the type `FldPadExact` (compare the name to the inexact `FldPad` type in Magma) which derives from `StrPadExact` (and so inherits its attributes) and has some additional attributes:

```
type FldPadExact[FldPadExactElt]: StrPadExact
attributes FldPadExact: xtype, prime, defining_polynomial
```

The `xtype` attribute takes one of the special enumerated values:

- `PRIME`: the field is $\mathbb{Q}_p$ for some p, and the `prime` attribute is p.
- `INERT`: the field is an unramified extension of another exact p-adic field K, and the `defining_polynomial` attribute is an inertial polynomial $f(x) \in K[x]$, defining the extension as $K(x)/(f(x))$.

- EISEN: the field is a totally ramified extension of another exact p-adic field K, and the defining_polynomial is an Eisenstein polynomial $f(x) \in K[x]$, defining the extension as $K(x)/(f(x))$.

The approximation field of an exact p-adic field is a corresponding inexact field, which in Magma has type FldPad. For the PRIME field $\mathbb{Q}_p$, this is simply pAdicField(p).

For extensions (INERT or EISEN) the situation is a little more complicated. In essence, we want the approximation to be the extension defined by an approximation of the defining_polynomial. The problem is that later we may want a more precise approximation, and so we have two choices:

- Replace the approximation field with a more precise approximation whenever it is required. This means that any element of the field may have an approximation lying in an older approximation field, and so will need to be coerced into the latest approximation field at some time.
- Use Magma's built-in semi-exact representation of p-adic extensions (§1.1): ext<K | m> where m is a map taking an integer and returning an approximation of the defining_polynomial to that precision.

We use the second choice because the explicit coercion between different approximation fields in the first choice was found to add a performance hit. It also has the benefit that we can talk of *the* approximation field, since it does not change in time.

Elements of exact p-adic fields are represented by the type FldPadExactElt. The meanings of its attributes are inherited from its parent type PadExactElt but to be explicit:

- parent is the FldPadExact field to which it belongs;
- approximation is an element of the approximation field of its parent, and is therefore a FldPadElt; and
- update is its update function, used to update the approximation.

We define coercion so that K ! <init, mkupdate> creates an element of K whose initial approximation is init and whose update function is mkupdate(x)

where `x` is the element being created, thus allowing the update function to refer to `x` itself:

```
intrinsic IsCoercible(K :: FldPadExact, args :: Tup)
    -> FldPadExactElt
  x := New(FldPadExactElt)
  x`parent := K
  x`init := args[1]
  x`update := args[2](x)
  return true, x
```

We provide intrinsics to access basic information; intrinsics for inertia degree and ramification degree are defined similarly:

```
intrinsic IsPrimeField(K :: FldPadExact) -> BoolElt
  return K`xtype eq PRIME

intrinsic DefiningPolynomial(K :: FldPadExact)
    -> RngUPolElt_FldPadExact
  if IsPrimeField(K) then
    error "not an extension"
  else
    return K`defining_polynomial

intrinsic BaseField(K :: FldPadExact) -> FldPadExact
  return BaseRing(DefiningPolynomial(K))

intrinsic Degree(K :: FldPadExact) -> RngIntElt
  return Degree(DefiningPolynomial(K))

intrinsic AbsoluteDegree(K :: FldPadExact) -> RngIntElt
  if IsPrimeField(K) then
    return 1
  else
    return Degree(K) * AbsoluteDegree(BaseField(K))
```

2.3 Univariate polynomials

A univariate polynomial ring over a p-adic field is represented by the type `RngUPol_FldPadExact` (analogous to the inexact type `RngUPolElt[FldPad]` in Magma) which also derives from `StrPadExact`:

```
type RngUPol_FldPadExact[RngUPolElt_FldPadExact]
attributes RngUPol_FldPadExact: base_ring
```

Such a ring is defined by its `base_ring`, an exact p-adic field (i.e. of type `FldPadExact`).

The `approximation` of such a ring must be the univariate `PolynomialRing` of the `approximation` of the `base_ring` (i.e. of type `RngUPol[FldPad]`).

2.4 The update function

The `update` attribute of a `PadExactElt` is a means to increase the precision of its `approximation` to a given absolute precision. Therefore it is natural to define it as a procedure which takes as input an absolute precision k, and whose side-effect is to replace the `approximation` by one whose precision is at least k. Using this definition will result in a working implementation of exact p-adics, but as we shall see in §3 it has some drawbacks and so in reality we use a slightly different definition. For now, however, it suffices to think of the update function in this way.

In the update function, instead of modifying the `approximation` of an element directly, one should use the following intrinsic which first checks that the update is consistent with the pre-existing approximation and in reality may perform more checks:

```
intrinsic Update(x :: FldPadExactElt, xx :: FldPadElt)
  assert IsWeaklyEqual(x`approximation, xx)
  x`approximation := xx
```

Instead of calling the `update` function directly, we should use the following intrinsic which ensures it is only called when required. In fact, since the `update` function is not a function at all then this intrinsic will actually have a different definition (§3.3), but with the same effect.

```
intrinsic IncreaseAbsolutePrecision(x :: PadExactElt, n)
  if not AbsolutePrecision(x) ge n then
    x`update(n)
```

In practice, we don't usually just want to increase the precision of an element, but we want the approximation itself. Hence we also make available an intrinsic `Approximation` to retrieve an approximation to an element to a certain absolute precision. It simply has to increase the absolute precision of the element, then return its approximation, perhaps with its precision decreased to the desired value.

```
intrinsic Approximation(x :: FldPadExactElt, n)
  IncreaseAbsolutePrecision(x, n)
  return ChangeAbsolutePrecision(x`approximation, n)
```

To increase the precision of an extension field, we just need to increase the precision of its `defining_polynomial` correspondingly. This ensures that the next time the semi-exact `approximation` field retrieves a defining polynomial, it will already be available to the given precision. There is nothing to be done for `PRIME` fields, since the prime is already represented exactly.

```
intrinsic IncreasePrecision(K :: FldPadExact, n)
  if not IsPrimeField(K) then
    IncreaseAbsolutePrecision(K`defining_polynomial, n)
```

The precision of a polynomial ring is the precision of its base ring, so to increase one we just have to increase the other:

```
intrinsic IncreasePrecision(R :: RngUPol_FldPadExact, n)
  IncreasePrecision(R`base_ring, n)
```

2.5 Examples

Example 2.1. Here is a definition of binary addition on two p-adic numbers. The initial approximation is simply the sum of the approximations of the inputs. The update function retrieves approximations to the inputs to the required precision, adds them, and sets this as the new approximation for the sum.

```
intrinsic '+' (x :: FldPadExactElt, y :: FldPadExactElt)
```

```
    -> FldPadExactElt
  init := x`approximation + y`approximation
  mkupdate := function (z)
    return procedure (n)
      Update(z, Approximation(x, n) + Approximation(y, n))
  return Parent(x) ! <init, mkupdate>
```

This example is an over-simplification compared to the true implementation in the following ways:

- The update function is not quite as described. See §3.
- The initial approximation `init` is computed to the current precision of the inputs, which may be overkill if they are both very precise. Instead, the implementation adds together approximations to "first precision", i.e.

  ```
  init := ChangeAbsolutePrecision(x`approximation,
    Min(WeakValuation(x)+1, AbsolutePrecision(x)))
    + ChangeAbsolutePrecision(y`approximation,
    Min(WeakValuation(y)+1, AbsolutePrecision(y)))
  ```

  As an optimization, most functions will compute the initial approximation from the inputs to first precision if possible.
- It should be checked that the inputs have the same `parent` field, or can be coerced to a common field. ◊

Example 2.2. Here we give a definition of binary multiplication, which is very similar to addition. The main change is in computing the precision required in the approximations.

```
intrinsic '*' (x :: FldPadExactElt, y :: FldPadExactElt)
    -> FldPadExactElt
  init := x`approximation * y`approximation
  mkupdate := function (z)
    return procedure (n)
      Update(z, Approximation(x, n - WeakValuation(y))
```

```
      * Approximation(y, n - WeakValuation(x)))
return Parent(x) ! <init, mkupdate>;
```
◊

3 ExactpAdics: Dependency tracking

3.1 Motivation

So far we have described a simple scheme for implementing exact p-adics, but it has drawbacks.

Example 3.1. Suppose we are given elements $a, b \in \mathbb{Q}_p$, and compute $c = a^3 + a^2b + ab^2 + b^3$, and wish to increase the absolute precision of c to 100.

We therefore require each of the summands a^3, a^2b, ab^2, b^3 to absolute precision 100. Now suppose that $\mathrm{val}(a) = 10$ and $\mathrm{val}(b) = 0$, so in fact we require the summands to relative precisions 70, 80, 90, 100 respectively. Hence we require a and b to these same relative precisions.

Therefore, if we increase the precision of each summand in turn to its required value, then we will be updating a first to relative precision 70, then 80, then 90 — i.e. absolute precisions 80, 90 and 100 — which is 3 separate updates. If updating a is an expensive operation, then this could become a performance issue.

Clearly, the right thing to do in this situation is to observe that we only need to update a once to absolute precision 100. With the current description of the `update` function, this is not possible. ◊

Our solution is to split updates into two steps: the first step identifies which other updates are required to occur first, we call these **dependencies**; the second step actually performs the update. With this explicit separation, we can find all of the dependencies of a calculation before satisfying any of them, allowing us to remove any redundancy as in the above example.

In the example, c has 4 dependencies, namely the 4 summands. Each of these summands in turn depends on one or both of a and b. There is redundancy in these dependencies because a and b each appear three times, and therefore could be merged.

3.2 Getters

We encapsulate these ideas into a new type[2]:

```
type Getter
attributes Getter: state, get_dependencies, get_value
```

A `Getter` represents an exact p-adic computation with dependencies. The `state` attribute is some getter-specific state which is passed by reference (and hence is modifiable) into the other functions.

The `get_dependencies` attribute is a `procedure(~state, ~deps)` which assigns to `deps` a list of dependencies. A dependency is a pair `<x,n>` where `x` is some p-adic value (i.e. a `PadExactElt`, such as a p-adic number or polynomial) and `n` is an absolute precision. Such a dependency should be interpreted as the getter saying "I can't compute my value until these values are to these absolute precisions." We say a dependency is **satisfied** if the absolute precision of `x` is at least `n`.

The `get_value` attribute is a `procedure(~state, ~value)` which, assuming that the dependencies previously reported are all satisfied, either assigns something to `value` or doesn't. If it does, then this is interpreted as the value of the computation. If it doesn't, then this is interpreted as the getter having more dependencies, and so `get_dependencies` needs to be called again.

Evaluating a getter means getting the value from the `get_value` procedure. Of course, this requires satisfying the dependencies reported by `get_dependencies` first, and leads to a recursive dependency satisfaction algorithm which we describe shortly.

3.3 Update function

With getters defined, we may now define precisely what an update function is: it is a function taking as input an absolute precision `n` and returning a `Getter`. This getter, when evaluated, will have the side-effect of increasing the absolute precision of the element to `n`. The value of the getter is ignored.

[2]In the package it is actually called `ExactpAdics_Gettr`

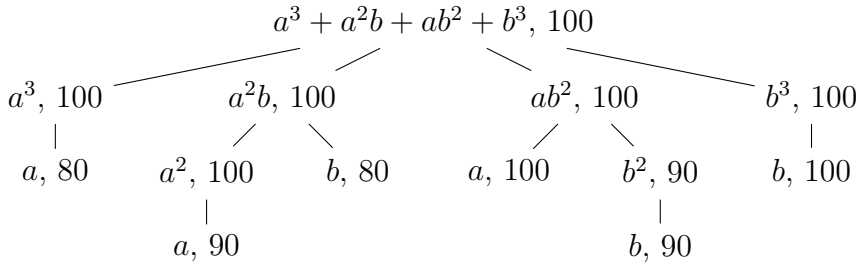

Figure 2: The tree of dependencies of the motivating example (Example 3.1).

With this definition, `IncreaseAbsolutePrecision` would actually be defined like so:

```
intrinsic IncreaseAbsolutePrecision(x :: PadExactElt, n)
  if not AbsolutePrecision(x) ge n then
    ignored := Evaluate(x`update(n))
```

Now to increase the absolute precision of a value, we just need to know how to evaluate a getter.

3.4 Evaluating getters

To evaluate a getter requires conceptually three steps: first we retrieve its dependencies from `get_dependencies`, then we satisfy those dependencies, then we retrieve the value via `get_value`. If `get_value` did not return a value, then we will need to repeat these steps.

Each of the dependencies is a pair `<x, n>` of a p-adic element and an absolute precision. Calling `x`update(n)` returns a getter which, on evaluation, increases the absolute precision of `x` to `n`, which we require. Hence we have reduced the problem of evaluating the original getter to the problem of evaluating these dependent getters. Recursing, we traverse the tree of dependencies all the way to its leaves. Figure 2 illustrates this dependency tree for the motivating example (assuming a and b themselves have no dependencies).

To avoid duplicated work, we want to combine all nodes for the same value together, taking the maximum of their absolute values, resulting in a directed acyclic graph such as in Figure 3.

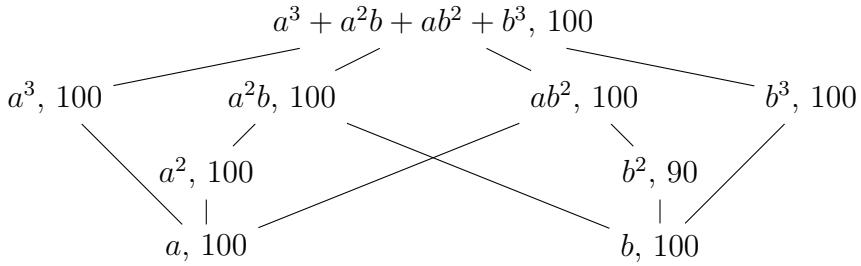

Figure 3: The merged graph of dependencies of the motivating example (Example 3.1).

A sink in this graph is precisely a getter with no dependencies. Therefore it may be evaluated and removed from the graph. Repeating, we will eventually reach the source, which can also be evaluated, and we have succeeded.

In practice, we do not represent these dependencies as a tree at all, but we take advantage of a simple fact: if the value `x` was created before value `y`, then `x` cannot possibly depend on `y`. We therefore keep track of the order of creation of elements by giving them a new attribute

```
attributes PadExactElt: id
```

to which we assign the value of a global counter when the element is created. We now represent the nodes of the graph simply as an associative array, where the node `<x,n>` is the value at the index `x`id`. Adding a new dependency into the tree is a matter of checking if there is already a dependency for this `x`; if so, then we should combine the old and new absolute precision in the array; if not, then we add the new node into the array. Traversing the tree in dependency order is now a matter of runnning through the indices of the array in sorted order.

We now present a version of this algorithm. The first procedure `add_dependencies` takes a list of dependency pairs `<x,n>` and recursively adds them and all their own dependencies into the array.

```
procedure add_dependencies(~array, todo_list)
  while #todo_list gt 0 do
    // pop an item from the todo list
    x, n := Pop(~todo_list)
```

```
    // if this is a new dependency, or the target
    // precision is greater than the existing one,
    // and it is not already satisfied, then replace
    // it and compute more dependencies to put in
    // the todo list
    if (x`id notin array or not n le array[x`id][2])
    and not n le AbsolutePrecision(x)
    then
      getter := x`update(n)
      array[x`id] := <x, n, getter>
      getter`get_dependencies(~getter`state, ~deps)
      for dep in deps do
        Append(~todo_list, dep)
```

The procedure `satisfy_dependencies` takes a list of dependency pairs and satisfies them all, first by calling `add_dependencies` to make an array of all dependencies, and then by running through the dependencies in order and trying to satisfy them. Here is one possible implementation:

```
procedure satisfy_dependencies(deps)
  // initially compute all dependencies
  array := AssociativeArray()
  add_dependencies(~array, deps)
  // keep trying until the graph is empty
  while #array gt 0 do
    // traverse the nodes in order
    for i in Sort(Keys(array)) do
      // do the update
      getter := array[i][3]
      getter`get_value(~getter`state, ~value)
      if assigned value then
        // success: remove the entry from the array
        delete array[i]
      else
```

```
      // failure: get more dependencies and start over
      getter`get_dependencies(~getter`state, ~deps)
      add_dependencies(~array, deps)
      break
```

The implementation in the `ExactpAdics` package behaves a little differently: if a particular node fails, then instead of immediately jumping back to the bottom of the tree, we continue traversing it to the top, but skipping over nodes which now have unsatisfied depdendencies. This is made possible by changing `add_dependencies` to explicitly track the children and parents of each node in the array, and altering `satisfy_dependencies` to:

```
procedure satisfy_dependencies(deps)
  // initially compute all dependencies
  array := AssociativeArray()
  add_dependencies(~array, deps)
  // keep trying until the graph is empty
  while #array gt 0 do
    // traverse the nodes in order
    for i in Sort(Keys(array)) do
      item := array[i]
      if item has no children then
        // do the update
        getter := item[3]
        getter`get_value(~getter`state, ~value)
        if assigned value then
          // success: remove the entry from the array
          for each parent of item do
            remove item as a child of parent
          delete array[i]
        else
          // failure: get more dependencies
          getter`get_dependencies(~getter`state, ~deps)
          add_dependencies(~array, deps)
```

Observe the main differences are that we now need to check a node has no children before processing it; when a node succeeds, we now need to remove it from the list of children of each of its parents; and when it fails, we no longer `break` out from looping over nodes.

Which of these two routines is better is arguable. The former routine may suffer from updating an early element of the graph, and then discovering later that the same element needs to be updated again, whereas the latter avoids this problem by performing as many updates on the graph as possible before starting over again. On the other hand, the latter routine may suffer from traversing the whole graph needlessely if an early node failed and is a child of everything else.

In practice, on problems of interest, we found that the latter routine usually performed better. That is, it was sometimes significantly faster and was rarely significantly slower, which is why the latter is used in the current implementation.

We also now define the intrinsic which evaluates a getter:

```
intrinsic Evaluate(g :: Getter)
  loop
    g`get_dependencies(~g`state, ~deps)
    satisfy_dependencies(~deps)
    g`get_value(~g`state, ~value)
    if assigned value then
      return value
```

3.5 Lazy computations

Now that we have a way of representing computations with dependencies, we now define some ways of combining and modifying them to produce more complex computations, with dependency tracking still built-in.

For instance, `Compose` takes as input a getter `g` and a function `f`, and returns a getter `h` such that `Evaluate(h) = f(Evaluate(g))`.

Similarly, `ComposeProcedure` takes as input a getter `g` and a procedure `f`, and returns a getter `h` such that `Evaluate(h)` has the same side-effects as calling `f(Evaluate(g))`.

Similarly, `ComposeGetter` takes as input a getter `g` and a function `f` returning a getter, and returns a getter `h` such that `Evaluate(h) = Evaluate(f(Evaluate(g)))`.

For added convenience, these compose functions can take a sequence of getters instead a single getter. In this case, the arity of the function `f` must equal the length of the sequence. For example `Evaluate(Compose([g1,g2],f)) = Evaluate(f(Evaluate(g1),Evaluate(g2)))`.

The intrinsic `Flatten` takes as input a sequence of getters, and returns the getter whose value is the sequence of values of the input getters.

There are also intrinsics for defining null getters, which do nothing, and for defining getters directly in terms of the `get_value` and `get_dependencies` functions. It is also possible to define getters directly whose dependencies are themselves getters, instead of `<x,n>` element-precision pairs.

The package also provides "lazy" versions of some intrinsics, which by convention are given the suffix `_Lazy`, which returns a getter which when evaluated has the same side-effects and return value as the non-lazy version.

For example the following intrinsic returns a getter `g` such that `Evaluate(g)` has the same side-effects as calling `IncreaseAbsolutePrecision(x, n)` directly:

```
intrinsic IncreaseAbsolutePrecision_Lazy
  (x :: PadExactElt, n) -> Gettr
  if AbsolutePrecision(x) ge n then
    return NullGetter()
  else
    return x`update(n)
```

Indeed, one could now define the non-lazy version as

```
intrinsic IncreaseAbsolutePrecision(x :: PadExactElt, n)
  ignored := Evaluate(IncreaseAbsolutePrecision_Lazy(x,n))
```

Similarly, the intrinsic `Approximation_Lazy(x, n)` returns a getter whose value is an approximation of `x` to absolute precision `n`.

Example 3.2. We can now present an implementation of binary addition using these tools. Compare this with the earlier version (Example 2.1), presented in

terms of the simplified representation where the update function was simply a procedure; now it is a getter, built using `ComposeProcedure` and `Approximation_Lazy` out of simpler getters.

```
intrinsic '+' (x :: FldPadExactElt, y :: FldPadExactElt)
    -> FldPadExactElt
  init := x`approximation + y`approximation
  mkupdate := function (z)
    return function (n)
      return ComposeProcedure(
        // lazily computes approximations to x and
        // y to precision n
        [ Approximation_Lazy(x, n)
        , Approximation_Lazy(y, n) ],
        // uses the approximations xx and yy to update
        // the value of z
        procedure (xx, yy)
          Update(z, xx + yy)
      )
  return Parent(x) ! <init, mkupdate>
```
◊

Most update functions in the package are defined in a similar fashion: firstly they lazily compute approximations to their inputs, and then they use these to update the value. The exceptions to this are mainly when the precision required of the input is not known immediately, and therefore some iteration is required; in such a circumstance, the getter returned by the update function usually needs to be defined directly in terms of its `get_dependencies` and `get_value` procedures.

4 `ExactpAdics`: Precision strategies

4.1 Motivating example

Suppose we want to compute the valuation of a p-adic number x. If the number is not currently weakly zero, then this is straighforward: the valuation is the weak valuation. Otherwise, we can't immediately deduce the valuation.

Therefore, we might try increasing the absolute precision of x. If the number is now not weakly zero, then we are done. Otherwise, we might increase the absolute precision further, and repeat this process for some time. At some point, if we don't discover the answer, we might give up.

How do we choose the amount to increase the absolute precision by? How long do we go for before giving up? A simple answer might be to keep doubling the precision forever until we succeed, but this would never terminate if $x = 0$.

On the other hand, the user may want the process to definitely terminate after some amount of effort, and therefore give up after the precision has reached some limit. Or, knowing more about the inputs, it may be more appropriate to increase the precision linearly instead of exponentially, for example. We abstract away such decisions into a precision strategy.

4.2 Definition

A **precision strategy** is a strictly increasing sequence of non-negative integers. The sequence may be finite or infinite in length.

4.3 Representation

How such a sequence is represented is not too important. In the `ExactpAdics` package, a precision strategy is represented as one of the following:

- A single non-negative integer n, which is a strategy of length 1: (n).
- A list of strategies, which is the contatenation of those strategies.
- A function m which takes an integer and either returns true and a larger value, or returns false. It represents a sequence of integers as follows: let n_0 be the previous value in the strategy; for $i = 0, 1, \ldots$, if $m(n_i)$ is false, then terminate the sequence, otherwise it is true and also returns n_{i+1}, the next element of the sequence.
- A string, which is interpreted as the global strategy with that name (see below).

- A tuple `<"limit",n>` which limits the remaining strategy to n; that is, it will terminate when the strategy reaches n. More precisely, the first time the strategy outputs a number $m \geq n$, it instead outputs n itself and then terminates.
- A tuple `<"exp",e>` which is equivalent to the function taking n to $\lceil n^e \rceil$. It therefore represents an infinite sequence which grows exponentially.
- A tuple `<"random">` which randomises the remaining strategy as follows: if at time i the previous value was n_i and the next value will be $n_{i+1} > n_i$, then we replace the next value with a uniform random element in $\{n_i + 1, \ldots, n_{i+1}\}$. This aims to dampen any potential issues arising from forcing precisions to come from a small set of values, such as powers of 2.

We provide the user with a global array of named strategies, with procedures to define and retrieve strategies with a particular name. Currently we define three named strategies by default:

- `"defaultLimit": <"limit", 100>`, not a precision strategy in itself, but can be mixed in to other strategies to limit them.
- `"unlimitedDefault": [1, <"randomize">, <"exp", 2>]`, starts at 1 and keeps doubling forever.
- `"default": ["defaultLimit", "unlimitedDefault"]`, the same as the previous strategy, but with the default limit applied.

4.4 Usage and conventions

Any function which makes a non-canonical decision about how to control the precision of its inputs should take one or more precision strategies as parameters to make this decision.

By convention, these parameters all have the word `Strategy` in their name, to make their purpose clear. Where there are any precision strategies parameters, there will be one with the name precisely `Strategy`. Its value is used as the default value for the others. Its default value is the string `"default"`, which refers to the global strategy with this name. For example:

```
intrinsic DoSomething( : Strategy := "default",
                       Strategy1 := Strategy,
                       Strategy2 := Strategy,
                       Strategy2b := Strategy2)
  ...
```

This means that for the typical user, it suffices to set a global strategy with the name `"default"` and then forget about precision strategies. If it then turns out that some computations are raising precision errors, the user can consider altering the precision strategy.

Using a precision strategy whose maximum value is 100 is functionally very similar to using inexact p-adics to precision 100. The difference is that in the inexact case, all computations are done to this precision, whereas in the exact case, if a computation can be done with less precision, then it will be.

4.5 Baseline precision

A function which takes a precision strategy parameter is free to use it in any fashion. It is intended, of course, that it will be interpreted as a sequence of precisions to try computations at, but there are different kinds of precision:

- Absolute precision: the integer k such that we know the value modulo π^k; that is, the approximation is of the form $x + \pi^k\mathcal{O}$.
- Relative precision: the absolute precision minus the weak valuation; that is, the integer r such that the approximation is of the form $\pi^v(x + \pi^r\mathcal{O})$.

If we interpret the entries of a precision strategy as absolute precisions, then it may be that the p-adic number actually has a large negative valuation, and so computing it to positive absolute precision is overkill. In a sense, the absolute precision is relative to the valuation 0, and 0 is an arbitrary choice.

If we interpret them as relative precisions, then we lose repeatability because the base-line for the relativeness can move: the weak valuation may increase in time. To demonstrate the issue, suppose we are computing the valuation of $x = 0$, initially known to absolute precision 10, and the precision strategy goes up to 100.

Interpreting the strategy as relative precisions, we would increase the absolute precision of x to 110. If we try to compute the valuation again, then we will increase the absolute precision again to 210. There is the potential to keep increasing the absolute precision of x indefinitely by repeatedly trying to compute its valuation.

We introduce a new kind of precision:

- Baseline precision: the absolute precision minus the "baseline valuation".

The **baseline valuation** is any fixed valuation attached to the value. Hence it may depend on the value, but does not change over time. By default, the baseline valuation is set to the weak valuation of the value when it is initially created. If the baseline valuation is set to 0, then we recover the absolute precision.

If we now interpret entries of the precision strategy as baseline precisions, then we avoid the two problems described above.

Example 4.1. Hence, a reasonable implementation of a function to compute the valuation is

```
intrinsic Valuation(x :: FldPadExactElt : Strategy:="default")
  for n in Strategy do
    IncreaseAbsolutePrecision(x, BaselineValuation(x) + n)
    if not IsWeaklyZero(x) then
      return WeakValuation(x)
  error "precision error"
```

◊

5 ExactpAdics2: Core structures and elements

5.1 Overview

Recall that in the `ExactpAdics` package, our updates are performed in terms of absolute precisions: an update function receives an absolute precision, and updates the approximation accordingly. In order to do this, the update function must compute the absolute precisions required of its dependencies, which are fed into the dependency-tracking framework, and recursively the update functions of the dependencies themselves must compute the absolute precisions required of their dependencies.

In the ExactpAdics2 package, we simplify this procedure by introducing a proxy for absolute precision. This proxy is a single positive integer n which we refer to as the **epoch**. At any given time, a p-adic object has a **current epoch** meaning that its current approximation is associated to that epoch. The precision of the current approximation must increase with the epoch.

Importantly, *by definition* the approximation of a p-adic object at epoch n depends only on the approximations of its dependencies at epoch n. Hence a p-adic object is represented by essentially two pieces of information: a list of the other p-adic objects on which it depends; and an **approximation function** which takes as input a list of approximations of its dependencies at some epoch n, and returns an approximation which is taken to be the approximation of the object at the same epoch.

Since the approximation function is only given the approximations of its dependencies at a given epoch n, all it must do is return an approximation to the best precision it can given its inputs. It is not aiming to return an approximation to any specific precision.

For example, our representation of $\mathbb{Q}_p$ has no dependencies, and its representation at epoch n is the fixed-precision field `pAdicField(p,2^n)` whose elements are of the form $\pi^v(y + \pi^r\mathbb{Z}_p)$ for $r \leq 2^n$. Hence the precision increases exponentially with n, the intention being that one will, in a small number of epochs, be able to increase the precision of a p-adic object to some desired absolute precision. Since there are only a small number of possible epochs — the user is highly unlikely to go beyond $n = 20$ — the dependency-tracking framework should only be invoked relatively infrequently.

As we shall see in §5.6, the dependency-tracking itself is also quite straightforward.

5.2 Abstract base types

As with ExactpAdics, this package uses the abstract types StrPadExact and PadExactElt to represent p-adic structures (such a fields and rings) and elements respectively. However, these are now also both subtypes of AnyPadExact, which represents any p-adic object:

```
type AnyPadExact
attributes AnyPadExact: id, dependencies,
  approximations, get_approximation

type StrPadExact: AnyPadExact

type PadExactElt: AnyPadExact
attributes PadExactElt: parent
```

The `id` attribute, as before, is a unique integer used to identify the object. It is assigned from a global counter, and so each object can only depend on objects with smaller `id`. This is used to simplify dependency tracking.

The `dependencies` attribute is a list of other p-adic objects (i.e. of type `AnyPadExact`) on which this one directly depends.

The `approximations` attribute is a list of approximations of the object. The object at position n in the list is the approximation of the object at epoch n. It is analogous to the `approximation` attribute in the `ExactpAdics` package, except that we now record all approximations.

The `get_approximation` attribute is the **approximation function**, and is analogous to the `update` function from the `ExactpAdics` package. It is a function with two inputs: an epoch (a positive integer) and the list of approximations of the `dependencies` at the given epoch. It must return the approximation of the object at the given epoch.

The `parent` attribute of an element (a `PadExactElt`) is the structure (a `StrPadExact`) containing the element. The approximation of an element at epoch n must be en element of the approximation of the parent at epoch n.

Figure 4 illustrates the relationships between these types and their attributes.

5.3 p-adic fields

The way in which p-adic fields and their elements are built on top of these base types is identical to the `ExactpAdics` package:

```
type FldPadExact[FldPadExactElt]: StrPadExact
attributes FldPadExact: xtype, prime, defining_polynomial
```

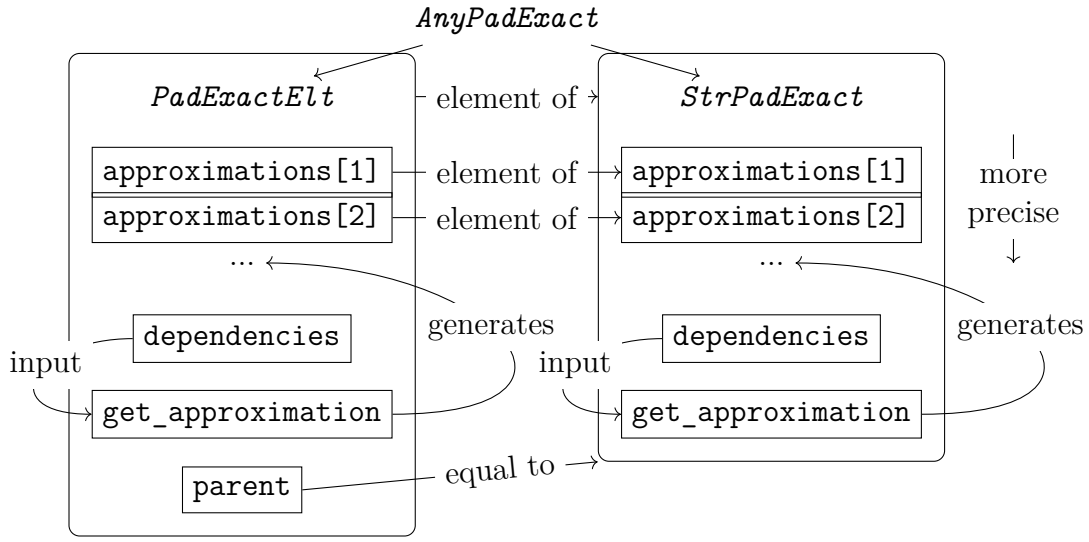

Figure 4: Illustration of the types `AnyPadExact`, `StrPadExact` and `PadExactElt`, their attributes and the relationships between them.

where the `xtype` is either `PRIME` indicating it is a prime p-adic field $\mathbb{Q}_p$, in which case the `prime` attribute must be set to the prime p, or else it is `INERT` or `EISEN` indicating an unramified or totally ramified extension, in which case the `defining_polynomial` attribute must be set to the inertial or Eisenstein defining polynomial.

The approximations of a p-adic field must be fixed-precision inexact p-adic fields, such as `pAdicField(2,20)` in Magma (representing $\mathbb{Q}_2$, whose elements have relative precision at most 20). However, two such fields in Magma are considered to be different, even if they only differ in their precision, and yet we will need to coerce approximate p-adic numbers between different approximate p-adic fields representing the same exact field, which will be manual and slow. Hence we define

attributes *FldPadExact*: `infinite_precision_approximation`

which is a semi-exact approximation of the field defined via a map, as described in §1.1. The `approximations` are then fixed-precision versions of this infinite-precision field produced via the `ChangePrecision` intrinsic in Magma. Since Magma now understands all of these approximations to come from a common

underlying field, it performs coercion between them for free.

5.4 Univariate polynomials

The way in which rings of univariate p-adic polynomials and their elements are defined is again identical to the `ExactpAdics` package:

```
type RngUPol_FldPadExact[RngUPolElt_FldPadExact]
attributes RngUPol_FldPadExact: base_ring
```

It is defined by its `base_ring`, a p-adic field (a `FldPadExact`). The approximation of such a ring at epoch n is the univariate polynomial ring over the approximation at epoch n of the base ring.

5.5 Examples

Example 5.1. Here we present a constructor for exact p-adic fields. It has no dependencies, and the precision of the approximation field is exponential in the epoch.

```
intrinsic ExactpAdicField(p :: RngIntElt)
    -> FldPadExact
  K := New(FldPadExact)
  K`dependencies := []
  K`get_approximation := function (n, xds)
    return pAdicField(p, 2^n)
  return K
```
◊

Example 5.2. Here we present an implementation of binary addition of p-adic numbers (cf. Example 3.2).

```
intrinsic '+' (x :: FldPadExactElt, y :: FldPadExactElt)
    -> FldPadExactElt
  z := New(FldPadExactElt)
  z`parent := x`parent
  z`dependencies := [x, y]
  z`get_approximation := function (n, xds)
```

```
    return xds[1] + xds[2]
  return z                                                    ◊
```

In fact, most purely arithmetic functions are this simple to implement.

Example 5.3. Here we present an implementation of polynomial resultant. Since the resultant depends on the degree of the polynomials, we need to ensure that the approximations of the inputs have the correct degree using `EnsureAllApproximationsAreFullDegree`, similar to as in Remark 5.7.

```
intrinsic Resultant (
  f :: RngUPolElt_FldPadExact,
  g :: RngUPolElt_FldPadExact)
    -> RngUPolElt_FldPadExact
  EnsureAllApproximationsAreFullDegree(f)
  EnsureAllApproximationsAreFullDegree(g)
  h := New(RngUPolElt_FldPadExact)
  h`parent := f`parent
  h`dependencies := [f, g]
  h`get_approximation := function (n, xds)
    return Resultant(xds[1], xds[2])
  return h                                                    ◊
```

5.6 Generating approximations

We now describe how we generate the `approximations` of a p-adic object from its `dependencies` and `get_approximation` function.

Suppose we are given a p-adic object and an epoch n, and we wish to compute the approximation of the object at the given epoch. The intrinsic `BringToEpoch` does this for us:

```
intrinsic BringToEpoch(x :: AnyPadExact, n :: RngIntElt)
  if #x`approximations lt n then
    for d in x`dependencies do
      BringToEpoch(d, n)
    xds := [d`approximations[n] : d in x`dependencies]
```

```
xx := x`get_approximation(n, xds)
x`approximations[n] := xx
```

First it checks if there is already an approximation at this epoch. If not, we run through the dependencies and bring these up to the same epoch recursively. Now we can construct a list of approximations of the dependencies at this epoch, pass this to `get_approximation` to produce the required approximation, and update the `approximations` list accordingly.

We also supply the intrinsic `EpochApproximation` which returns the approximation at a given epoch:

```
intrinsic EpochApproximation(x :: AnyPadExact, n :: RngIntElt)
  BringToEpoch(x, n)
  return x`approximations[n]
```

The true implementation of `BringToEpoch` is slightly more complicated. The following subsections explain how.

Saving the approximation

Instead of simply saving the output of `get_approximation` as a new approximation directly, we assign it using a generic intrinsic called `SetApproximation`, which performs some checks. This includes checking that the approximation is of the right type; that, if it is an element, the approximation is an element of the approximation of its parent; and that the approximation is consistent with the current best approximation attached to the object.

Furthermore, our package assumes that if an object has an approximation at epoch n, then it has approximations for all lower epochs. So what do we do if we are jumping from epoch 1 to 10, for example, how do we set the intermediate approximations? For each subtype of `AnyPadExact`, there must be an intrinsic `InterpolateEpochs` implemented which takes as input a p-adic object, a range of epochs, and an approximation at the top epoch. It must return a list of approximations for the intermediate epochs.

Example 5.4. The default implementation uses the approximation function to generate the intermediate values, provided we exceed the `min_epoch`. This is

usually sufficient for structures. Note that the dependencies are guaranteed to be at the top epoch already.

```
intrinsic InterpolateEpochs(x :: AnyPadExact,
    n1 :: RngIntElt, n2 :: RngIntElt, xx :: FldPadElt)
  if n1 ge x`min_epoch then
    return [x`get_approximation(n, xds)
      where xds := [d`approximations[n] : d in x`dependencies]
      : n in [n1..n2-1]]
  else
    error "not implemented: InterpolateEpochs with min_epoch>1"
```

◊

Example 5.5. For p-adic numbers, we coerce the approximation into the approximations of the parent field at the intermediate epochs.

```
intrinsic InterpolateEpochs(x :: FldPadExactElt,
    n1 :: RngIntElt, n2 :: RngIntElt, xx :: FldPadElt)
  return [x`parent`approximations[n] ! xx : n in [n1..n2-1]]
```

◊

Now if `SetApproximation` is setting some approximation for a high epoch, it will use `InterpolateEpochs` to fill in the gaps.

Minimum epoch

Example 5.6. Suppose we are implementing division of two p-adic numbers. Here is what looks like a reasonable implementation:

```
intrinsic '/' (x :: FldPadExactElt, y :: FldPadExactElt)
  require IsDefinitelyNonzero(y)
  z := New(FldPadExact)
  z`parent := x`parent
  z`dependencies := [x, y]
  z`get_approximation := function (n, xds)
    return xds[1] / xds[2]
  return z
```

Note, however, that even though we checked that `y` is non-zero, we are not guaranteed that all of its approximations are not weakly zero. If some of them are, then the division inside `get_approximation` may raise an error. ◊

To solve this, we have `IsDefinitelyNonzero` return a second value, which is the smallest epoch at which an approximation for `y` is not weakly zero. Note that since approximations may not become less precise as epoch increases, this implies that all approximations for `y` are not weakly zero above this epoch. We can then set the new attribute

```
attributes AnyPadExact: min_epoch
```

to this epoch.

The meaning of `min_epoch` is that it is the smallest epoch for which the `get_approximation` function should be called, and hence in our example, the division will only use non weakly zero approximations to `y`.

To use `min_epoch`, we simply need to insert the following line into `BringToEpoch`

```
n := Max(n, x`min_epoch)
```

which ensures that the epoch we are updating to is at least `min_epoch`.

Remark 5.7. For the particular case of division, and similar functions, we can take a different approach and implement it like so:

```
intrinsic '/' (x :: FldPadExactElt, y :: FldPadExactElt)
  EnsureAllApproximationsAreNonzero(y)
  z := New(FldPadExactElt)
  z`parent = x`parent
  z`dependencies := [x, y]
  z`get_approximation := function (n, xds)
    return xds[1] / xds[2]
  return z
```

where, as the name suggests, the intrinsic `EnsureAllApproximationsAreNonzero` ensures that all approximations of `y` are not weakly zero. This is achieved by first calling `IsDefinitelyNonzero` to check `y` is nonzero and find an epoch at which

its approximation is not weakly zero, and then by using `InterpolateEpochs` and `SetEpochs` to interpolate this approximation down to epoch 1.

Maximum epoch

Analogous to `min_epoch`, there is also

```
attributes AnyPadExact: max_epoch
```

which is the maximum epoch at which `get_approximation` should be called.

The intention here is that the user can set the maximum epoch on a p-adic object as a way of limiting the precision to which computations involving that object are performed.

The `BringToEpoch` intrinsic is modified to insert a check that the target epoch is not greater than the `max_epoch`, if it is set. If so, it will raise a precision error:

```
if assigned x`max_epoch and n gt x`max_epoch then
  error "precision error: max_epoch exceeded"
```

We also supply the intrinsic `CanBringToEpoch` which is the same as `BringToEpoch` except that instead of raising a precision error when the `max_epoch` is reached it returns false, and otherwise returns true to signal success.

5.7 Precision strategies

At present, we do not provide functionality analogous to the precision strategies (§4) of the `ExactpAdics` package. The only method for controlling precision currently available to the user is the `max_epoch` attribute described in §5.6, which will cause an error to be raised if a computation requires too much precision.

Therefore currently, any functions which need to increase the precision of its inputs do so simply by trying each epoch in order. Hence, there are no `Strategy` parameters in this package, and instead we can think of the sequence $1, 2, \ldots$ as the default strategy where the values are now epochs, not precisions.

Example 5.8. `Valuation` is implemented like this (cf. Example 4.1):

```
intrinsic Valuation(x :: FldPadExactElt) -> Val_FldPadElt
  for n in 1,2,... do
```

```
BringToEpoch(x, n)
if not IsWeaklyZero(x) then
  return WeakValuation(x)
```
◊

6 Comparison of `ExactpAdics` and `ExactpAdics2`

6.1 Complexity of updates

Compare the procedures `satisfy_dependencies` (§3.4) of `ExactpAdics` and `BringToEpoch` (§5.6) of `ExactpAdics2`, which are the underlying means in each package of generating an approximation to a p-adic object.

In the latter, we satisfy each dependency recursively immediately. In the former, we perform a backwards pass to gather all dependencies together, followed by a forwards pass to satisfy dependencies.

The rationale for the behaviour of the former was discussed in §3.1, and it comes down to the fact that the same p-adic object may appear multiple times in a dependency with different absolute precisions. By performing the backwards pass first, we can merge all such dependencies into one. On the other hand, in `BringToEpoch` all dependencies are being brought to the same epoch, and therefore we can satisfy each dependency immediately without risk of it needing to be brought to a higher epoch later.

Additionally, in `ExactpAdics`, satisfying a dependency is allowed to fail (i.e. the `get_value` procedure of a `Getter` is allowed to not return a value), which triggers a new backwards pass to find dependencies of this failed update, and an extra forwards pass will have to occur to satisfy these. Hence there is in principle no bound on the amount of dependency tracking required to update a single element, whereas in `ExactpAdics2` we do a single pass.

This is a necessary feature of the design of `ExactpAdics`: because the `update` function must update its target object to a given absolute precision, we must allow it the freedom to take a guess at the precision required of its dependencies, and then try a better guess if it turns out this was too low. This is because there are some operations where it is difficult or impossible to determine the dependency precisions in advance.

On the other hand, in ExactpAdics2, because there is a looser relationship between precisions and epochs, the get_approximation function is not aiming for any specific precision. Instead, it simply needs to produce an approximation to the best precision it can.

6.2 Number of updates

In ExactpAdics2, an "update" can occur at each epoch. Since precisions are exponential in the epoch (recall §5.1 and Example 5.1), then typically there are only a small number of epochs ever considered, rarely going beyond epoch 20. This limits the number of times the dependency tracking framework ever needs to consider a single object, and so the time spent doing dependency tracking is essentially a small constant times the number of variables in a computation.

On the other hand, in ExactpAdics one can in principle increase the precision of an element by 1 many times, and each time the dependency tracking code will be invoked, so there is essentially no bound on the time spent doing this. To mitigate this, one could modify the package so that elements can only increase their precisions by large jumps, such as doubling each time.

6.3 Implementing new functions

To implement a new low-level operation in ExactpAdics, such as addition of two p-adic numbers, requires implementing a Getter which (a) can compute the precisions to which its dependencies are required; and (b) compute an approximation, given approximations of its dependencies. To do the same in ExactpAdics2 only requires (b), and therefore implementing new functionality in the latter is often much quicker.

Furthermore, actually computing the dependency precisions can be slow:

Example 6.1. Let $h(x) = f(x)g(x)$ be a product of two polynomials. Suppose we want to compute an approximation to h with the absolute precision of the kth coefficient (h_k) at least a_k. Then we need f_i to absolute precision $\max_j a_{i+j} - \mathrm{val}(g_j)$ and g_j to absolute precision $\max_i a_{i+j} - \mathrm{val}(f_i)$.

Computing these absolute precisions is of the same order of complexity as performing the multiplication itself. On the other hand, the multiplication is implemented in a low level compiled language such as C, whereas our `ExactpAdics` package is implemented in the high-level interpreted language Magma, and so computing these absolute precisions can be far more expensive. ◊

6.4 Precision optimality

By design, all computations in `ExactpAdics` are performed to as little precision as is possible to get the answer. In `ExactpAdics2`, we perform all computations starting from the same initial precision and keep doubling this precision as necessary. Hence the latter is not optimal in terms of precision used, but is typically within a factor of 2 of optimal.

It is possible for `ExactpAdics2` to be worse than this. Suppose x is cheap to compute approximations for, but loses a lot of precision along the way, so if it has an approximation in `pAdicField(p,2^n)` then its precision is significantly less than 2^n. Also suppose that y is expensive to compute, and does not lose any precision. Let $z = x + y$. Now because x loses a lot of precision, so does z, and therefore computing an approximation to z requires a relatively high epoch. Computing the approximation to y at this epoch is expensive, but also unnecessary because it achieves this required precision at an earlier epoch.

In a sense, the epoch is not really a proxy for the precision of an element, but a proxy for the worst precision of all the dependencies of the element.

For most common applications, the amount of precision lost tends to be bounded and small as epoch increases, and so this effect is minimal.

6.5 Precomputing dependencies

In §9.1 we describe a generic optimization technique which can make updating the approximations of a selected p-adic object much quicker. This is done by pre-computing some of its dependency graph so that it can be traversed more efficiently.

This optimization opportunity is only possible in `ExactpAdics2`. Even if `ExactpAdics` were redesigned to make the list of dependencies of an object explicit,

so that a piece of the dependency graph could be precomputed, we would still need to do a backwards pass to find the minimal precision required of each dependency.

6.6 Precision strategies

In `ExactpAdics`, whenever a function needs to increase the precision of an object in a non-canonical way, it does so according to a precision strategy (§4), giving fine control over each precision tried.

On the other hand, `ExactpAdics2` currently has no such functionality other than setting the `max_epoch` parameter on an object (§5.7). When a function needs to increase the precision of an object in a non-canonical way, it repeatedly increases the epoch by 1. In practice, precision strategies in `ExactpAdics` will usually just repeatedly double the precision, which behaviour is almost the same as increasing the epoch by 1 in `ExactpAdics2`. In principle, the package could have strategies to control which epochs are used, but this is not yet implemented.

6.7 Timings

Dependency tracking

In this section, we describe an experiment designed to stretch the dependency tracking capabilities of our packages. This involves performing a computation which involves thousands of intermediate variables, but the steps themselves are cheap to compute.

In this experiment, we define $x_1 = 1, x_2 = 2 \in \mathbb{Q}_2$ and for $i = 3, \ldots, 10000$ we define $x_i = x_{j_i} + x_{k_i}$ for some randomly chosen $j_i, k_i \in \{1, 2, \ldots, i-1\}$. Finally we define $y = \sum_{i=1}^{10000} x_i$. We time how long it takes to compute y to absolute precision 2^n for $n = 1, \ldots, 16$, taking the total time — this emulates a typical sequence of increasing the absolute precision of y 16 times. We repeat this 10 times with different random choices and take the mean.

This experiment is repeated using a number of different p-adic implementations, with the mean timings given in Table 1. Note that the random choices are made in advance and so are not timed, and we use the same random seed in each experiment, so precisely the same sequence of operations is being compared.

Experiment	Time (sec)				
(1) Builtin	0.949				
(2) `ExactpAdics`	174.284	=	6.907	+	167.377
(3) `ExactpAdics2`	7.330	=	0.400	+	6.930
(4) `ExactpAdics2` (opt: default)	7.047	=	0.528	+	6.519
(5) `ExactpAdics2` (opt: fast)	2.006	=	0.460	+	1.546

Table 1: Timings for a highly dependent computation over different implementations, including two optimizations.

Experiment (1) uses the builtin inexact p-adics available in Magma, and so is a reasonable lower bound on what we can expect to achieve. Experiments (2) and (3) use the `ExactpAdics` and `ExactpAdics2` packages, respectively. These timings are broken into two parts, the first part being the time to construct y, and the second part being the time to increase its precision to $2, 4, \ldots, 2^{16}$. We can see that the latter package outperforms the former significantly on both counts.

Experiments (4) and (5) are the same as (3), except we use the optimization techniques described in §9.1 to make y directly depend only on x_1 and x_2. Experiment (4) uses the default version, which gives a small speed-up. Experiment (5) uses the "fast" version, which forgets the intermediate variables and uses the `get_approximation` functions directly, and achieves a significant speed-up.

Real-world example

We compute the 2-part of the conductor of the hyperelliptic curve

$$C : y^2 = -2x^6 - 15x^4 - 37x^2 - 30$$

using our implementation of [18] mentioned in §1. This implementation can use either of our packages for its underlying p-adic computations. The curve has discriminant $\Delta = -2^{16} \cdot 3 \cdot 5$ and conductor $N = 2^{10} \cdot 3 \cdot 5$.

When using `ExactpAdics`, this takes 146 seconds, compared to 33 seconds for `ExactpAdics2`. The time spent in the dependency-tracking portion of code, which includes actually computing approximations, is 124 and 25 seconds respectively, with 22 and 12 seconds respectively left over to other computations.

With `ExactpAdics`, this 124 seconds spent in dependency tracking is divided equally between generating approximations and tracking dependencies (this includes calling the update function and computing dependencies). About half of the latter is spent computing dependencies, most of the rest being logic comparing absolute precisions.

In fact, of the whole 146 seconds, 39 seconds is spent just constructing our representation of a valuation of a univariate polynomial. Individually this is fast, but we construct 240,000 of them throughout the algorithm. This demonstrates the benefit of using epochs instead of fine absolute precisions in `ExactpAdics2`.

Note that the number of times the dependency tracking framework is invoked is about 54,000 and 40,000 for the two packages. Given the same algorithms are used in both packages, we expect these numbers to be similar. In this case we do not appear to have the potential issue that the framework is invoked too often. The number of p-adic objects created is about 11,000 and 7,000 for the two packages.

6.8 Conclusions

Given the above arguments and evidence, we currently recommend the typical user to choose `ExactpAdics2` over `ExactpAdics`.

On the other hand, if more of the internal workings were implemented at a lower level than the Magma language and optimized, then it may be that `ExactpAdics` could be made comparably fast. Indeed, much of the comparative slowness in `ExactpAdics` comes from the need for a lot of simple arithmetic to compute absolute precisions, which is typically slow in an interpreted language such as Magma.

7 Additional structures

So far we have described the representation of p-adic numbers and univariate polynomials over p-adic fields. We now briefly describe two more structures provided by the package.

7.1 Multivariate polynomials

A multivariate polynomial ring over a p-adic field is represented by the type `RngMPol_FldPadExact` (analogous to the inexact type `RngMPolElt` in Magma) which derives from `StrPadExact`:

```
type RngMPol_FldPadExact[RngMPolElt_FldPadExact]
attributes RngMPol_FldPadExact: base_ring, rank
```

Such a ring is defined by its `base_ring`, an exact p-adic field (i.e. of type `FldPadExact`), and by its `rank`, the number of indeterminates.

An approximation of such a ring must be the multivariate `PolynomialRing` of an approximation of the `base_ring` of the same rank.

7.2 Cartesian products

The cartesian product of a number of exact p-adic structures is itself an exact p-adic structure, and has the type `SetCart_PadExact` analogous to the type `SetCart` for general cartesian products.

```
type SetCart_PadExact[Tup_PadExact]
attributes SetCart_PadExact: components
```

Such a cartesian product is defined by its `components`, a list of exact p-adic structures.

An approximation of this structure must be the cartesian product (a `SetCart`) of an approximations of its components.

Why do we define this specialised form of cartesian products, when a general one exists already? The difference is that a standard tuple of exact p-adic values treats the component values as completely independent objects, whereas the exact tuple links them together in the sense that they have a single common update/approximation function. Therefore, the exact tuple is an appropriate choice for a collection of p-adic values which belong to some conceptually higher structure.

Example 7.1. Suppose we wish to implement a Hensel-lifting routine which takes as input a sequence $F \in K[x_1, \ldots, x_n]^n$ of n multivariate polynomials of rank n

over some p-adic field K and a sequence $X \in K^n$ of n elements of K such that we can apply Hensel's lemma to deduce there is a root $Y \in K^n$ of F close to X, and returns the sequence Y (as in §9.9).

To update the components of Y we perform a Hensel-lifting routine which is essentially some $n \times n$ linear algebra depending on F and X, the important point being that this computes all components of Y to some precision simultaneously; it is not possible to compute one component of Y to high precision in isolation. Therefore it makes sense to represent Y as a tuple with a single update function.

By comparison, if we represented Y as a sequence of independent values, then each component would still need to maintain its own approximation to the whole vector Y in order to perform Hensel lifting. Worse still, increasing the precision on one component would perform Hensel lifting, but then only update that one component even though the information is available to update all components. Therefore, increasing the precision of all components of Y would be n times too slow. ◊

8 Valuations

In our packages, the valuation of a p-adic element `PadExactElt` is intended to be the finest measure available of the valuation of the components of the element. Because there are many different types of p-adic elements (e.g. numbers, polynomials, tuples), there are as many different types of valuations, all needing to be represented somehow. There are some operations common to all valuations, such as addition, so we define a new abstract type to represent all types of valuation:

```
type Val_PadExactElt
```

and we shall later define sub-types corresponding to each p-adic structure.

Note that the difference of two valuations is also a valuation, corresponding to the division of two p-adic elements with those valuations. Therefore, all kinds of precisions — absolute, relative and baseline — are also valuations.

In the packages, we use valuations of subtype of `Val_PadExactElt` to represent all valuations (including weak valuations) and all precisions. In particular, the

input to an update function is a valuation in this form, representing the intended absolute precision.

attributes *Val_PadExactElt*: value

The `value` field of a valuation contains the actual value of the valuation, whose representation is element-dependent.

8.1 Valuations of p-adic numbers

Ordinarily we think of the valuation of a p-adic number as an integer, except that:

- The valuation of zero is not an integer, it takes the special value ∞.
- Multiplication of two valuations is not a useful concept: it has no description in terms of the arithmetic of p-adic numbers.
- On the other hand addition and infimum do make sense: the valuation of the product of p-adic numbers is the sum of their valuations, and the valuation of their sum is lower-bounded by the minimum of their valuations.
- Multiplication and division of a valuation by an integer or rational number also does make sense, since it corresponds to exponentiation of a p-adic number.
- It is useful to be able to talk about the valuation of elements in an extension, and these may be rational numbers.
- Subtraction is useful to define, since a relative or baseline precision is the difference of two valuations. In particular, we also need to include the symbol $-\infty := 0 - \infty$.
- Supremum is also a useful operation: if we increase the absolute precision of a p-adic number several times, then its final absolute value is the maximum of the intermediate absolute precisions.

We deduce that the standard ring of integers $(\mathbb{Z}, +, \times)$ is not a useful structure for valuations to reside in; instead, we define the set $Z := \mathbb{Q} \cup \{\pm\infty\}$, elements of which we represent with the type:

`type` *`Val_FldPadElt`*`:` *`Val_PadExactElt`*

The `value` attribute is either an integer (a `RngIntElt` in Magma), a rational number (a `FldRatElt`) or $\pm\infty$ (a `Infty`).

The following operations are supported:

- Addition: Defined for all pairs of elements of Z, except $\infty + (-\infty)$ is left undefined and will cause an error.
- Subtraction: Defined for all pairs of elements of Z. In particular, $\infty - \infty$ is defined to be 0; this is because the p-adic number 0 represented to infinite p-adic absolute precision has infinite weak valuation, and so $\infty - \infty$ should be its relative precision, which is 0. While an arbitrary collection of additions and subtractions is not associative by these definitions, in practice if subtraction is only used to compute precisions, then the results will be well-defined.
- Infimum and supremum (which are the operations `meet` and `join` in the Magma language).
- Multiplication and division by rational numbers (which we term **scaling**).
- Equality, inequality, and orderings $=, \neq, \leq, <, \geq, >$. In particular, Z is totally ordered.
- An operation called `diff` which is defined as follows: `x diff y` is `x` if `x > y` and otherwise is $-\infty$. Note that it is the lowest valuation `z` such that `z join y = x join y`. It has a natural interpretation in our context: if we require an element to have precision `x` and its current precision is `y` then `x diff y` is the lowest valuation `z` such that increasing the precision to `z` suffices. Whilst defining such an operation for single p-adic numbers may seem like overkill, it turns out to be useful for aggregates.

8.2 Valuations of aggregate structures

All other p-adic structures in the package are aggregate structures, in the sense that they represent, perhaps recursively, a collection of p-adic numbers. As defined

at the top of the section, a valuation in the package is the finest possible description of the components of a p-adic element, and therefore we represent valuations of an aggregate as an analogous aggregate of valuations. Specifically:

- Univariate polynomials: A polynomial $f(x) = \sum_{i=0}^{\infty} f_i x^i \in K[x]$ over a p-adic field K may be more simply thought of as the infinite sequence $(f_0, f_1, \ldots)$ of its coefficients, which is zero for all but finitely many places. Correspondingly, its valuation we represent as the infinite sequence $(\mathrm{val}(f_0), \mathrm{val}(f_1), \ldots)$, which is ∞ at all but finitely many places. If we subtract two such valuations pointwise, the result is an infinite sequence which is 0 at all but finitely many places. Most generally then, a valuation of a univariate polynomial is an infinite sequence which takes the same value at all but finitely many places.

  In the package, we define the new type `AssocDflt` which represents an associative array with a default value; that is, it has a default value so that if a key is not in the array, then the value of the array at that key is the default. These are useful for representing functions which are constant at all but finitely many places.

  Valuations of univariate polynomials are represented by the type:

  type *Val_RngUPolElt_FldPad*: *Val_PadExactElt*

  whose `value` is a default associative array `AssocDflt` whose keys are non-negative integers i and whose values are `Val_FldPadElt`s.

- Multivariate polynomials: A polynomial

$$f(x_1, \ldots, x_r) = \sum_{e \in \{0,1,\ldots\}^r} f_e x_1^{e_1} \cdots x_r^{e_r} \in K[x_1, \ldots, x_r]$$

  of rank r over K can be thought of as the map $e \mapsto f_e$ taking exponent vectors to the corresponding coefficient. As with univariate polynomials, this map is zero almost everywhere. In analogue with univariate polynomials, we represent the valuation of a multivariate polynomial with the type:

  type *Val_RngMPolElt_FldPad*: *Val_PadExactElt*

whose `value` is a default associative array `AssocDflt` whose keys are exponent vectors e and whose values are the corresponding `Val_FldPadElt`s.

- Tuples: Valuations of tuples `Tup_PadExactElt` of exact p-adic elements are represented by the type:

  type `Val_Tup_PadExactElt:` *`Val_PadExactElt`*

  whose value is a corresponding tuple of valuations, representing the valuations of the components of the tuple.

These valuations all support the following operations:

- Addition, subtraction, scaling, infimum (`meet`), supremum (`join`), `diff`: These are all defined point-wise.

- Equality and inequality: two valuations are equal iff they are equal point-wise.

- Ordering: two valuations are ordered if that ordering applies point-wise.

Note that while the set $Z = \mathbb{Q} \cup \{\pm\infty\}$ of valuations for p-adic numbers is totally ordered — and this ordering is respected by the ordering, infimum and supremum operations — the valuations for aggregate p-adic elements are only partially ordered. For example two tuples in $\mathbb{Q}_2^2$ may have valuations $(1, 2)$ and $(2, 1)$ and so are not ordered relative to each other, or they may have valuations $(1, 2) < (2, 2)$. This partial ordering is respected by infimum and supremum; for example `x join y` is the unique smallest valuation greater than or equal to both `x` and `y`.

Example 8.1. Suppose a univariate polynomial of degree 5 is known to absolute precision $x = (3, 5, 8, 10, 13, 2, \infty, \infty, \ldots)$. The infinite precisions indicate that we know that coefficients 6 upwards are precisely zero. Also suppose we want to increase its absolute precision to at least $y = (10, 10, \ldots)$. Then it suffices to

increase it to

$$\begin{aligned} y \texttt{ diff } x &= (10 \texttt{ diff } 3, 10 \texttt{ diff } 5, 10 \texttt{ diff } 8, 10 \texttt{ diff } 10, \\ &\qquad 10 \texttt{ diff } 13, 10 \texttt{ diff } 2, 10 \texttt{ diff } \infty, \ldots) \\ &= (10, 10, 10, -\infty, -\infty, 10, -\infty, \ldots) \end{aligned}$$

and so we see it suffices to only increase the precisions of coefficients 0, 1, 2 and 5. ◊

9 Additional features

We now describe some of the high-level features available in our packages, including notes on how they are implemented. The majority of these features are in both packages, but any pseudo-code in this section will be as in `ExactpAdics2`.

9.1 Precomputing dependencies

Suppose $d = (d_1, \ldots, d_k)$ are p-adic objects, and x is some complicated expression in d, such as in §6.7. Hence x does not depend directly on d, it depends on some intermediate expressions which recursively ultimately depend on just d. To compute an approximation for x requires traversing its dependency graph, including all these intermediate expressions, which will be time-consuming. If we do not care about the intermediate expressions, then it could be more efficient to compute approximations to x directly from d.

In the `ExactpAdics2` package, we provide an intrinsic `WithDependencies` which takes a p-adic object x and a list d of other p-adic objects and returns a copy of x whose direct `dependencies` are precisely d. The basic idea is that we pre-compute the piece of the dependency graph between x and d, which the `get_approximation` function can traverse efficiently.

Specifically, starting from x, we recursively traverse its dependencies, gathering them together to form the set of all of its dependencies. Whenever we reach a dependency lying in d, we terminate that branch of the recursion, so that we only find the dependencies between x and d. Next, we sort these dependencies by `id`

into a list. Since the `id`s are assigned sequentially, this also sorts according to dependency.

With its default behaviour, `WithDependencies` also incorporates information about the `min_epoch` of each dependency into this list: specifically the list is now a list of pairs (y, m) where m is the maximum `min_epoch` of y or anything depending on y. Having precomputed this list, we can define `get_approximation` to traverse this list in order: given an epoch n, for each (y, m) in the list, we compute an approximation to y at epoch $\max(n, m)$ from its dependencies, which will already be at this epoch, and update y accordingly.

`WithDependencies` also has a `Fast` parameter which performs a more aggressive optimization. Note that the default behaviour still explicitly deals with all intermediate dependencies (y, m), and in particular each such y is updated in the usual manner, which involves a number of consistency checks. The "fast" version ultimately forgets these dependencies entirely and instead just remembers the `get_approximation` function attached to each one. These are called directly, one by one, with the approximations they return just appended to a temporary list, which is used as input when calling the next one, and so on. The last item in this list will be the approximation to `x` returned by `get_approximation`. As a result, there is no cacheing or consistency checking of each intermediate approximation, which can be a significant speed-up. The final answer, which is used to update x, is still checked in the usual manner so we do not lose any safety.

Note that because the "fast" algorithm does not allow cacheing of intermediate variables, the `min_epoch` of the created object must be the maximum of the `min_epoch`s of all dependencies. Similarly its `max_epoch` must be the minimum of those of its dependencies.

Furthermore, the "fast" algorithm assumes that the `approximations` of the intermediate variables are all as produced by `get_approximation`. Therefore division, which can change the approximations of its dependencies (Remark 5.7), should not be an intermediate expression. For this reason, the `Fast` parameter is false by default because it is not guaranteed to be safe. We also give division (and other intrinsics with the same issue) a `Safe` parameter which, when true, does not use this trick and is therefore safe to be an intermediate expression, at the cost of a potentially higher `min_epoch`.

Timings demonstrating the benefits of using this optimization technique are given in §6.7. As noted in §6.5, this generic optimization is not possible in `ExactpAdics`.

9.2 Valuation comparison

A common p-adic operation is to compare the valuation of a p-adic number x with some given valuation v. Consider the following code:

```
if Valuation(x) gt 10 then
  ...
```

The first thing this does is compute the valuation of x precisely, and then compare the answer with 10. However, this is overkill: since there is no canonical way to increase the precision of x in order to find its valuation (which may be very high), then `Valuation` will proceed according to some precision strategy, and therefore could never return an answer, or could raise a precision error.

We provide the following intrinsic:

```
intrinsic ValuationGe(x, n)
  IncreaseAbsolutePrecision(x, n)
  return WeakValuation(x) ge n
```

so that `ValuationGe(x, n)` is functionally very similar to `Valuation(x) ge n` except that now there is a canonical way to increase the precision of x in order to get the answer, and it is guaranteed to produce a result with as little precision as required.

In reality, the definition of `ValuationGe` is made a little more complex by checking if the answer is already known without increasing the precision of x.

We similarly provide analogues `ValuationEq`, `ValuationNe`, `ValuationLt`, `ValuationGe` and `ValuationGt` for the other comparison operators.

9.3 Residue class fields and higher quotients

Since Magma's inexact p-adics includes some functionality around residue class fields, we make similar functionality available in our package.

The intrinsic function `ResidueClassField` takes as input an exact p-adic field K (type `FldPadExact`) and returns its residue class field $\mathbb{F}$ (type `FldFin`) and the quotient map $q : \mathcal{O} \to \mathbb{F}$.

This is implemented by computing the residue class field of the `approximation` field of K (i.e. `ResidueClassField(K`approximation)`), which returns $\mathbb{F}$ and the quotient map $\tilde{q} : \tilde{\mathcal{O}} \to \mathbb{F}$ where $\tilde{\mathcal{O}}$ is the integer ring of the `approximation` field. Then q may be defined in terms of $\tilde{q}$: given $x \in K$, increase the absolute precision of x to at least 1, and then call $\tilde{q}(\tilde{x})$.

The quotient map $\tilde{q}$ also comes with a partial inverse, an embedding $\tilde{q}^{-1} : \mathbb{F} \hookrightarrow \tilde{\mathcal{O}}$, which we similarly extend to a partial inverse $q^{-1} : \mathbb{F} \hookrightarrow \mathcal{O}$. In this case, $q^{-1}(x)$ is always given to absolute precision 1, and cannot have its absolute precision increased; in a sense, it refuses to choose among the many possible pre-images. In order to force such a choice, the intrinsic `WeakApproximation` is provided, which takes as input an exact p-adic number, and returns another exact p-adic number which is equal to the input up to the precision of the input.

In a completely analogous manner, the intrinsic `Quotient(K, n)` returns the ring $\mathcal{O}/\pi^n\mathcal{O}$ and the quotient map q, which again has a partial inverse. Hence `Quotient(K, 1)` and `ResidueClassField(K)` are equivalent, except that the latter represents the result as a field, and not a more general ring.

9.4 Completions of number fields

Magma's inexact p-adics includes some functionality around taking completions of number fields at finite primes, so we make similar functionality available in our package.

The procedure `ExactCompletion` takes as input a number field F and a finite place $\mathfrak{p}$ of F, and returns the completion $K := F_\mathfrak{p}$ as an exact $\mathfrak{p}$-adic field, and the embedding map $e : F \hookrightarrow K$.

This is implemented around the builtin intrinsic `Completion` which takes the same inputs, and returns the completion $\tilde{K}$ as a semi-exact p-adic field, and the embedding $\tilde{e} : F \hookrightarrow \tilde{K}$. Then K is simply an exact p-adic field whose approximation is $\tilde{K}$, and $e : F \hookrightarrow K$ returns an element whose update function uses $\tilde{e}$ to embed the input element of F into $\tilde{K}$ to sufficiently high precision.

9.5 Newton polygons

The following definitions and results are standard, if not the notation.

Definition 9.1. If $f(x) = \sum_{i=0}^{d} f_i x^i \in K[x]$ is a polynomial over a p-adic field K, then its **Newton polygon** $\mathcal{N}(f)$ is the lower convex hull in $\mathbb{Q} \times \mathbb{Q}$ of the points $(i, \mathrm{val}(f_i))$. It can also be interpreted as the graph of a function $[0, d] \to \mathbb{Q}$, also denoted by $\mathcal{N}(f)$. By definition, this function is continuous, convex and piece-wise linear. If $\mathcal{F}$ is a face of the Newton polygon, i.e. a line segment from (i_0, v_0) to (i_1, v_1), then its **width** is $w(\mathcal{F}) = w = i_1 - i_0$ and its **slope** is $s(\mathcal{F}) = \frac{v_1 - v_0}{i_1 - i_0}$. Writing $s(\mathcal{F}) = -\frac{h}{e}$ in lowest terms, then the **ramification degree** of the face is $e(\mathcal{F}) = e$, and the **residual polynomial** is $r(\mathcal{F})(x) = \sum_{i=0}^{w/e} \overline{f_{ie+i_0} \pi^{ih - v_0}} \in \mathbb{F}_K[x]$.

Lemma 9.2. *If $\mathcal{F}$ is a face of $\mathcal{N}(f)$, then f has precisely $w(\mathcal{F})$ roots in K^{alg} of valuation $-s(\mathcal{F})$. Writing $-s(\mathcal{F}) = h/e$ in lowest terms, if r is such a root, then $r^e \pi^{-h}$ has valuation 0 and $r(\mathcal{F})(\overline{r^e \pi^{-h}}) = 0$. Furthermore the roots r of valuation h/e are in e-to-1 correspondence with roots (possibly repeated) of $r(\mathcal{F})(x)$ via $r \mapsto \overline{r^e \pi^{-h}}$.*

Hence the Newton polygon and related quantities provide much information about the roots of a polynomial, and so are invaluable in scenarios such as root-finding or factorization of polynomials.

We provide an intrinsic

```
intrinsic NewtonPolygon(f :: RngUPolElt_FldPadExact
  : Support:=<0,Degree(f)>)
    -> NwtnPgon
```

which takes as input a p-adic polynomial `f` and returns its Newton polygon. Since computing this involves computing the valuations of some of its coefficients, which may initially be weakly zero, it takes a `Strategy` parameter. It also takes a `Support` parameter which is a pair of integers representing a range, and the returned value will be a sub-polygon of the full Newton polygon supported on at least this range; this can be useful if, for example, the polygon might have a single root at 0, and so it suffices to get the piece of the Newton polygon on $[1, \infty)$.

The Newton polygon is computed as follows. We loop through precisions in the `Strategy` and for each one, compute a corresponding approximation `xf` of `f`.

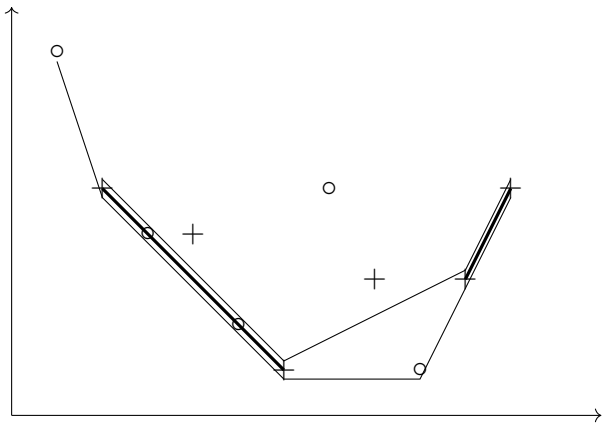

Figure 5: Computation of a section of a Newton polygon (heavy line) from lower and upper weak Newton polygons. Circles indicate the weak valuations of weakly zero coefficients, crosses indicate valuations of non weakly zero coefficients. Observe that since each end of the leftmost piece of the Newton polygon is at a vertex of the lower polygon, then these must also be vertices of the Newton polygon; contrast with the rightmost piece, in which the face could extend further to the left.

We compute the **lower weak Newton polygon** of `xf`, defined to be the lower convex hull of the points (i, w_i) where w_i is the weak valuation of the ith coefficient of `xf`. We also compute the **upper weak Newton polygon** of `xf`, defined to be the lower convex hull of the points (i, w_i) such that the ith coefficient of `xf` is not weakly zero, and therefore $w_i = \mathrm{val}(f_i)$. The lower weak Newton polygon lies below the Newton polygon, which in turn lies below the upper weak Newton polygon. Therefore if the weak polygons overlap anywhere, then that overlap is a section of the Newton polygon (see Figure 5). If this section includes all of the `Support` then we are done, otherwise we move on to the next precision in the strategy.

9.6 Ramification polygons and transition functions

The ramification filtration of $\mathrm{Gal}(L/K)$, the Hasse-Herbrand transition function and the upper-numbering of ramification groups are all standard, and appear for instance in Serre [60, Ch. IV]. The theory extends to non-Galois extensions [35], which we summarise now.

Definition 9.3. Given a finite extension L/K of p-adic fields, its **Galois set** $\Gamma(L/K)$ is the set of K-embeddings of L into a normal closure — this is a generalization of the Galois group. For $\sigma \in \Gamma$, we define $\mathrm{val}(\sigma) := \min_{x \in \mathcal{O}_L} \mathrm{val}_L(\sigma x - x)$ and $\Gamma_v := \{\sigma \in \Gamma : \mathrm{val}(\sigma) \geq v\}$ for $v \geq 0$. The **(lower) ramification breaks of** L/K are the v at which the function $v \mapsto |\Gamma_v|$ is discontinuous. We define the **transition function**

$$\phi_{L/K}(v) = \frac{1}{e(L/K)} \int_0^v |\Gamma_t| \, dt$$

which is continuous, piecewise linear, increasing and hence bijective $[0, \infty) \to [0, \infty)$, and letting $\psi_{L/K}$ be its inverse, we define $\Gamma^u = \Gamma_{\psi(u)}$. This defines the **upper ramification numbering**. We define $L^u = L_v$ to be the fixed field of $\Gamma^u = \Gamma_v$ (where $u = \phi(v)$).

The following lemma summarizes some key apsects of the theory. In particular, the upper numbering is well-behaved under changing the top field and fixing the base field, much in the way that the lower numbering is well-behaved under changing the base field. It also shows that the Galois correspondence generalizes to the sets Γ^u.

Lemma 9.4 ([35, Prop. 2, Rmk. 3, Prop. 3])**.**

(a) *If $M/L/K$ then $\phi_{M/K} = \phi_{L/K} \circ \phi_{M/L}$.*

(b) *Also $\Gamma^u_{L/K} = \{\sigma|_L : \sigma \in \Gamma^u_{M/K}\}$, and in particular $\Gamma^u_{L/K}$ are restrictions of elements of* $\mathrm{Gal}(L/K)^u$.

(c) *$(L : L^u) = |\Gamma^u|$ and so in particular L^u is the subfield of L fixed by* $\mathrm{Gal}(L/K)^u$.

Computing quantities such as the transition function and upper/lower ramification breaks of an extension L/K is therefore of use when considering the Galois action of inertia or higher ramification groups. To compute these, we use ramification polygons, detailed decriptions of which appear in [32, §4–5] and [53, §3]. We summarize the key points here.

Definition 9.5. Suppose U/K is unramified, degree d, $f(x) \in U[x]$ is Eisenstein degree e, defining the totally ramified L/U, with uniformizer $\pi \in L$ such that

$f(\pi) = 0$. Then the **ramification polygon of** L/K is the Newton polygon of the polynomial $f(x + \pi)$ (which is supported on $[1, e]$) with an additional horizontal face supported on $[e, ed]$.

Lemma 9.6. *The lower ramification breaks of L/K are v where $-v$ is a slope of a face of the ramification polygon. The corresponding $|\Gamma_v|$ is the abscissa of the right hand vertex of the corresponding face. Letting $v_0 = 0 < \ldots < v_t$ be the lower breaks in sorted order and $s_i = |\Gamma_{v_i}|$, and letting $u_0 = 0 < \ldots < u_t$ be the upper breaks (i.e. $u_i = \phi_{L/K}(v_i)$) then*

$$\frac{u_{i+1} - u_i}{v_{i+1} - v_i} = \frac{s_i}{e(L/K)}$$

gives a means to compute any one of these three sequences from the other two.

Proof. Since $\mathcal{O}_L = \mathcal{O}_U[\pi]$, for $\sigma \in \Gamma(L/U) = \Gamma(L/K)_1$ we have $\mathrm{val}(\sigma) = \mathrm{val}(\sigma(\pi) - \pi) > 0$. Now $\sigma(\pi) - \pi$ are precisely the roots of $f(x + \pi)$, and so by Lemma 9.2 their valuations correspond to faces of the ramification polygon. Specifically, if $-v$ is the slope of the face and w its width, then there are w elements $\sigma \in \Gamma(L/U)$ such that $\mathrm{val}(\sigma) = v$. Accumulating these widths from the left gives the sizes of Γ_v, as claimed, for $v > 0$. The extra horizontal face by construction has slope 0 and vertex at $ed = (L : K) = |\Gamma| = |\Gamma_0|$. The formula relating v_i, s_i, u_i follows from the definition of $\phi_{L/K}$ as an integral. □

Hence the slopes and abscissa of vertices of faces of the Newton polygon correspond to (v_i, s_i) and the vertices of the transition function correspond to (v_i, u_i), and there is a bijective correspondence between these sequences. Therefore we can compute transition functions from Newton polygons and vice versa, provided we represent the transition function by its vertices. We introduce a new type to do so:

```
type HasseHerbTransFunc
attributes HasseHerbTransFunc: vertices
```

It is easy to evaluate the transition function at a given v or its inverse at u by interpolating between the vertices. If we have the transition functions $\phi_{L/K}$ and $\phi_{M/L}$, then $\phi_{M/K} = \phi_{L/K} \circ \phi_{M/L}$ has as its lower breaks the union of: (a) the lower

breaks of $\phi_{M/L}$; and (b) $\phi_{L/K}^{-1}$ applied to the lower breaks of $\phi_{L/K}$. The upper breaks are similar, and hence we have the vertices of $\phi_{M/K}$ and therefore deduce a function to compose transition functions.

Now if we are given such an $M/L/K$ say, with M/L and L/K each defined by an Eisenstein polynomial over an unramified extension, then we can compute the ramification polygons of M/L and L/K via the definition above. From this, we can compute the transition functions $\phi_{M/L}$ and $\phi_{L/K}$. From these and the composition routine described above, we can compute $\phi_{M/K}$ and from this compute the ramification polygon of M/K. In this manner, we deduce an intrinsic `RamificationPolygon` to compute the ramification polygon of an arbitrary extension of p-adic fields and `TransitionFunction` to compute the corresponding transition function.

9.7 Hensel's lemma for univariate root-finding

Recall Hensel's classic lemma.

Lemma 9.7 (Hensel)**.** *Suppose $f(x) \in \mathcal{O}[x]$, $a \in \mathcal{O}$ such that $v(f(a)) \geq s > 0 = v(f'(a))$. Then there exists a unique $b \in K$ such that $f(b) = 0$ and $v(a-b) \geq s$. More precisely, defining $a' := a - f(a)/f'(a)$ then $v(f(a')) \geq 2s$ and $v(f'(a')) = 0$, so iterating $a \mapsto a'$ then $a \to b$.*

We refer to the iteration process in Hensel's lemma as "Hensel lifting". It can be generalized to non-integral inputs:

Lemma 9.8. *Suppose $f(x) \in K[x]$, where K is a p-adic field, and $a \in K$ such that among all roots b of f, $v(a-b)$ is maximised precisely once. Then iterating $a \mapsto a - f(a)/f'(a)$ yields $a \to b$.*

Proof. The generalization is actually reducible to the original version.

Consider the polynomial $f(x+a)$. Its roots are $b-a$ where b is a root of f, and so its Newton polygon measures the number of times each $v(a-b)$ occurs. Hence the hypothesis is equivalent to saying that the first face of the Newton polygon of $f(x+a)$ has width 1.

Suppose this is true, then in particular the first face has integral slope and so there exist $j, k \in \mathbb{Z}$ so that $g(x) := \pi^j f(\pi^k x + a)$ has integral coefficients,

$\mathrm{val}(g_0) > 0$ and $\mathrm{val}(g_1) = 0$. Note that $g_0 = g(0)$ and $g_1 = g'(0)$ so the original version of Hensel's lemma applies to g and 0. By linearity, Hensel lifting on g is equivalent to Hensel lifting on f. □

Remark 9.9. Krasner's lemma is a corollary of this form of Hensel's lemma.

We provide an intrinsic `IsHenselLiftable` which takes as input a polynomial $f(x) \in K[x]$ and an element $a \in K$ and returns true if this generalized version of Hensel's lemma can be applied to find a root b of f close to a. If so, it also returns that root.

The algorithm proceeds by computing $f(x + a)$ to sufficient precision to see if the first face of its Newton polygon has width 1 or not. If so, then the returned root has as its initial approximation the approximation of a truncated to a certain precision determined by Hensel's lemma, and its update function performs the Hensel lifting iteration above.

```
intrinsic IsHenselLiftable(
  f :: RngUPolElt_FldPadExact,
  a :: FldPadExactElt)
    -> BoolElt, FldPadExactElt

  // first determine if Hensel's lemma is applicable
  // try successively precise approximations
  for n in 1,2,... do
    // get an approximation of f and a
    xf := EpochApproximation(f, n)
    xa := EpochApproximation(a, n)
    // approximate f(x+a)
    xf2 := Evaluate(xf, x + xa)
    // this Newton polygon is computed from the *weak*
    // valuations, so is not necessarily correct
    np := NewtonPolygon(xf2)
    face := Faces(np)[1]
    // if the first face has width 1 and the right hand
    // vertex is correct, then there really is a face of
```

```
    // width 1
    if Width(face) eq 1
    and not IsWeaklyZero(Coefficient(xf, 1))
    then break
    // if the face has higher width, and both vertices
    // are correct, then there really is a face of this
    // width
    elif Width(face) ne 1
    and not IsWeaklyZero(Coefficient(xf, 0))
    and not IsWeaklyZero(Coefficient(xf, EndVertices(face)[2][1]))
    then return false
    // else we cannot conclude whether the first face
    // has width 1 or not
    else continue
  // if we get this far, then a is Hensel liftable
  // we omit the implementation of Hensel lifting
  root := ...
  return true, root
```

9.8 Univariate root finding I

Magma provides an intrinsic `Roots` to find all of the roots of a univariate polynomial over an inexact p-adic field. As discussed in §1.2, perhaps confusingly these are roots "up to precision", so for example given the polynomial $x^2 + 2^{10}\mathbb{Z}_2$ over $\mathbb{Q}_2$, it will return the root $0 + 2^{10}\mathbb{Z}_2$ with multiplicity 2. In a sense this is misleading, because it could be that the polynomial is actually $x^2 + 2^{11}$ to absolute precision 10, and this polynomial does not have any roots. Hence, one should not interpret the existence of roots of an inexact polynomial to necessarily be roots of any lift of that polynomial to something more precise.

On the other hand, a `Roots` intrinsic for exact polynomials should only return genuine roots of the full-precision polynomial. We can use the inexact `Roots` intrinsic and `IsHenselLiftable` to achieve the desired result:

```
intrinsic Roots(f :: RngUPolElt_FldPadExact) -> []
```

```
for n in 1,2,... do
  // get an approximation to f
  xf := EpochApproximation(f, n)
  // compute the roots of f up to precision
  xroots := Roots(xf)
  // check that the roots are all Hensel liftable
  roots := []
  for xroot in xroots do
    // the roots must be distinct, up to precision,
    // to have a chance of succeeding; if not, go
    // to the next precision in the strategy
    if Multiplicity(xroot) ne 1 then
      continue n
    // see if an approximation to the root is
    // Hensel liftable to a genuine root of f
    ok, root := IsHenselLiftable(f, xroot)
    // if not, then go to the next precision
    if not ok then
      continue n
    // if we get this far, we have a root
    Append(~roots, root)
  // if we get this far, we have a full set of roots
  return roots
// if we get this far, we have run out of things to try
error "precision error"
```

Note that this can only succeed if all of the roots over the base field are simple, because Hensel's lemma can only detect simple roots. This is the best possible: if f has a root r of multiplicity m, then to any precision this is indistinguishable from f having an irreducible factor of degree m, all of whose roots are very close to r. For example, over $\mathbb{Q}_2$, the root 1 to multiplicity m is indistinguishable to high precision from an irreducible factor whose roots are $1 + 2^{10000}\sqrt[m]{2}$. Hence it is not possible to prove that a polynomial to any finite precision has repeated roots.

9.9 Hensel's lemma for multivariate root finding

We are now interested in solving square systems of multivariate polynomials, namely we wish to find the roots of systems of n polynomials $f(x) = (f_1(x), \ldots, f_n(x)) \in K[x]^n$ in n variables $x = (x_1, \ldots, x_n)$. A root of such a system is an element $r \in K^n$ such that $f(r) = 0$.

The following multivariate version of Hensel's lemma is well-known:

Lemma 9.10. *Suppose $f(x) \in \mathcal{O}[x]^n$ is a system of n polynomials in n variables, $a \in \mathcal{O}^n$, $\operatorname{val}(f(a)) \geq s > 2t = 2\operatorname{val}(\det J(f)(a))$ where $J(f)_{i,j} = \frac{df_i}{dx_j}$. Then there is a unique $b \in K^n$ so that $f(b) = 0$ and $\operatorname{val}(a-b) \geq s-t$. More precisely, defining $a' = a - f(a)J(f)(a)^{-1}$, then $\operatorname{val}(\det J(f)(a')) = t$ and $v(f(a')) \geq 2(s-t)$; therefore iterating $a \mapsto a'$ then $a \to b$.*

We can state a slightly more general version, which says that if we can apply a linear change to the equations, perhaps over an extension, such that Hensel's lemma applies, then Hensel's lemma also applies to the original system over the base field:

Lemma 9.11. *Suppose $f(x) \in K[x]$ is a system of n polynomials in n variables, $a \in K^n$, L/K a finite extension, $M, N \in \mathrm{GL}_n(L)$, $\tilde{a} := Ma \in \mathcal{O}_L^n$, $\tilde{f} := Nf(M^{-1}x) \in \mathcal{O}_L[x]^n$, $\operatorname{val}(\tilde{f}(\tilde{a})) \geq s > 2t = \operatorname{val}(\det J(\tilde{f})(\tilde{a}))$. Then a Hensel lifts to a unique root of f in K.*

Proof. Define $\tilde{a} = Ma$, $\tilde{a}' = \tilde{a} - \tilde{f}(\tilde{a})J(\tilde{f})(\tilde{a})^{-1}$. We know that iterating $\tilde{a} \mapsto \tilde{a}'$ then $\tilde{a} \to \tilde{b} \in L$ a root of $\tilde{f}$. By linearity we find $\tilde{a}' = Ma'$ where $a' = a - f(a)J(f)(a)^{-1}$. We conclude that $a \to b$ such that $Mb = \tilde{b}$, and since $a' \in K$, then $b \in K$ also. □

In the package, we provide an intrinsic `IsHenselLiftable` which takes as input such a system f and a near-root a and returns true if Hensel's lemma is applicable. If so, it also returns the Hensel-lifted root itself. It also optionally accepts two vectors $\mu, \nu \in \mathbb{Q}^n$ which define the diagonal matrices M and N with diagonal entries π^μ and π^ν, and uses the more general version of Hensel's lemma. This allows us to implicitly rescale the equations and variables, so that the inputs need not be integral.

It should be possible to determine whether there exists any such μ and ν so that Hensel's lemma is applicable, and therefore recover a completely general and parameterless version of multivariate `IsHenselLiftable` in analogue with the univariate case. The theory for this has not been completely worked out yet.

Remark 9.12. An algorithm to actually compute the roots or factors of such a square system is work in progress (see Chapter VI).

9.10 Hensel's lemma for univariate factorization

Suppose $f(x) \in K[x]$ is a monic univariate polynomial of degree $n = n_1 + n_2$. Consider the problem of finding a factorization $f(x) = g(x)h(x)$ where $\deg(g) = n_1$, $\deg(h) = n_2$ and g and h are monic. By treating the n coefficients of $1, x, \ldots, x^{n-1}$ in $f(x) - g(x)h(x)$ as multivariate polynomials in the n_1 coefficients of g and the n_2 coefficients of h, we have a system of n multivariate polynomials in n variables to solve.

We conclude that there is a version of Hensel's lemma applicable to this situation, provided that a given near-factorization $f(x) \approx g(x)h(x)$ is sufficiently accurate. How accurate this needs to be is controlled by the determinant of the Jacobian matrix J in Hensel's lemma. In this case, the first n_1 rows of J correspond to $\frac{d(f-gh)}{dg_i} = x^i h(x)$, and the next n_2 rows correspond to $\frac{d(f-gh)}{dh_i} = x^i g(x)$, with the columns being the coefficients of these polynomials. This is precisely the matrix defining the resultant, and so we conclude that $\det(J) = \mathrm{Res}(g, h)$.

For example, we get the following version of Hensel's lemma for factorization, although more general versions analogous to those in previous sections are also possible.

Lemma 9.13. *Suppose $f(x), g(x), h(x) \in \mathcal{O}[x]$ are monic of degrees $n = n_1 + n_2, n_1, n_2$ such that $\mathrm{val}(f - gh) \geq s > 2t = 2\,\mathrm{val}(\mathrm{Res}(g, h))$. Then g, h Hensel-lift uniquely to a factorization of f.*

Suppose we are given $f(x)$ and $g(x)$ but not $h(x)$ and want to determine if $g(x)$ is Hensel liftable to a factor of $f(x)$. It seems natural to define $h := f \operatorname{div} g$ and apply Hensel's lemma to this. The following lemma shows that this is indeed the best choice for h:

Lemma 9.14. *If $f(x), g(x) \in K[x]$ have degrees n and $n_1 \leq n$ and g is monic, then among polynomials $h(x) \in K[x]$ of degree $n_2 = n - n_1$, $\mathrm{val}(f - gh)$ is maximized by $h = f \operatorname{div} g$.*

Proof. By definition, $f - g(f \operatorname{div} g) = f \bmod g =: h_0$. Consider arbitrary $h = f \operatorname{div} g + d$, then $f - gh = f - g(f \operatorname{div} g) - gd = h_0 - gd$. Define $B = \mathrm{val}(h_0) + 1$ and suppose there exists d so that $\mathrm{val}(h_0) - gd \geq B$. In particular $d \neq 0$. Fix d of smallest degree, and let m be this degree. Then the $(m + n_1)$th coefficient of $f - gh$ is $-d_m$ and so $\mathrm{val}(d_m) \geq B$. Define $d' = d - d_m x^m$, then $\mathrm{val}(h_0) - gd' \geq B$ and $\deg d' < \deg d$, a contradiction. □

The package provides an intrinsic `IsHenselLiftable` which takes as input two polynomials f and g and returns true if g is Hensel-liftable to a factor of f. If so, it also returns the factor itself. In analogue with the multivariate version of `IsHenselLiftable`, this intrinsic takes parameters which implicitly re-scale the polynomials and the variable x before applying Hensel's lemma.

9.11 Univariate factorization by Newton polygon

An easy application of Hensel's lemma for univariate factorizations is to factor a polynomial according to its Newton polygon.

Recall that the slopes of faces of the Newton polygon of a polynomial $f(x)$ correspond to valuations of roots of $f(x)$, with the width of the face corresponding to the number of roots with this valuation. If two roots of $f(x)$ come from the same irreducible factor, then they are Galois conjugate and so have the same valuation; we conclude that each face of the Newton polygon corresponds to a factor of f whose degree is the width of the face.

In fact, we can prove this fact directly using a version of Hensel's lemma for factoring, seen in the previous section: it is not hard to see that with a suitable choice of rescaling on f and x that we may choose g so that Hensel's lemma is applicable. Specifically, we rescale so that the selected face of the Newton polygon of f becomes horizontal and incident with the x-axis, and take for g the polynomial formed from the coefficients of f corresponding to the face.

The package provides a routine `NewtonPolygonFactorization` which takes as input a univariate polynomial f and returns its factorization according to

its Newton polygon. It is implemented essentially by first computing the `NewtonPolygon` of f, and then for each face constructing a suitable g and calling `IsHenselLiftable` to produce a factor.

9.12 Univariate factorization into irreducibles I

We also provide an intrinsic `Factorization` which returns the full factorization of a polynomial $f(x)$ into irreducible factors.

It is implemented in a very similar fashion to `Roots` as described in §9.8: it calls Magma's builtin `Factorization` routine on an approximation to $f(x)$, and then checks if each factor returned is Hensel liftable using `IsHenselLiftable`.

9.13 Univariate root finding and factorization into irreducibles II

Our `Roots` and `Factorization` intrinsics actually have a parameter `Alg` to select between two different algorithms. We have already described `Alg:="Builtin"` (§9.8, §9.12) which is a wrapper around the builtin intrinsics for inexact p-adics.

With the parameter `Alg:="OM"`, which is now the default, we use our own implementation of an "OM algorithm" for computing "Okutsu invariants" of the input polynomial, which identifies its irreducible factors and some properties of the extensions they define. From these, we can use "single factor lifting" to generate arbitrarily precise approximations to the factors. The algorithm is essentially that described in [62, Ch. VI].

Remark 9.15. Although not usually presented as such, "single factor lifting" is nothing but Hensel's lemma in disguise. Recall in 9.10 that we expressed factoring $f(x) = g(x)h(x)$ as a multivariate system of equations whose coefficients are the $n_1 = \deg(g)$ coefficients of g and the $n_2 = \deg(h)$ coefficients of h.

We can instead write $g(x) = x^{n_1} + \sum_{i<n_1} g'_i X_{g,i}(x)$ and $h(x) = x^{n_2} + \sum_{i<n_2} h'_i X_{h,i}(x)$ where $X_{*,i}(x) \in K[x]$ are fixed monic polynomials of degree i, and instead consider $f(x) = g(x)h(x)$ as a system of equations in the variables g'_i and h'_i. Essentially, we have chosen bases for the vector spaces of monic polynomials of degrees n_1 and n_2 different from the usual $1, x, x^2, \ldots$. This is a linear change

of variables of the sort considered in Lemma 9.11.

The OM algorithm builds up such a basis for each factor, and the point in the algorithm at which an irreducible factor is identified is precisely the point at which Hensel's lemma, in terms of this basis, can be invoked.

Remark 9.16. The same algorithm is also made available as an intrinsic `Exactp-Adics_Factorization` which can take an inexact p-adic polynomial. This can be used independently of the package.

Chapter V

Conductors of Genus 2 Curves

Foreword

This chapter has been published separately as an article [18] co-authored with Tim Dokchitser. Sections 4 to 6 (except 4.1) represent my contribution, they are mainly to do with the wild conductor exponent and the implementation. The remaining sections are by Tim Dokchitser, they are mainly to do with the tame conductor exponent.

1 Introduction

One of the main arithmetic invariants of a curve $C/\mathbb{Q}$ (or over a number field) is its **conductor**. It is a representation-theoretic quantity measuring the arithmetic complexity of C, and it is particularly important in the considerations that involve Galois representations or L-functions of curves.

In practice, the conductor is difficult to compute. It is defined as a product $N = \prod_p p^{n_p}$ over primes p, so the problem is computing the local **conductor exponents** n_p; these are functions of $C/\mathbb{Q}_p$. For elliptic curves (genus 1), the problem of computing n_p is solved with Tate's algorithm [66] and Ogg-Saito formula [50, 58]. In genus 2 and $p \neq 2$ there is an algorithm of Liu [44] via the Namikawa–Ueno classification [49], and for hyperelliptic curves of arbitrary genus there is a formula for the conductor [19], again for $p \neq 2$.

As the global conductor N requires the knowledge of n_p for *all* primes p, including $p = 2$, it is currently only provably computable for elliptic curves, and for quotients of modular curves using modular methods (see e.g. [29]). In practice, one can guess N from the functional equation of the L-function (see e.g. [14, 7]), but this approach is conditional on the conjectural analytic continuation of the L-function, and is basically restricted to reasonably small N.

In this paper, we propose an (unconditional) algorithm to compute the conductor for curves of genus 2. The case to consider is $p = 2$, so from now on C will be a non-singular projective curve of genus 2, defined over a finite extension K of $\mathbb{Q}_2$. Recall that the conductor exponent is the sum of the **tame** and **wild** parts (see §2),

$$n_2 = n = n_{\text{tame}} + n_{\text{wild}}.$$

The difficult one is the wild part, which is the Swan conductor of the l-adic Tate module of the Jacobian J/K of C/K, for any $l \neq 2$. We will take $l = 3$ and use that n_{wild} can be computed from the action of $\text{Gal}(\bar{K}/K)$ on the 3-torsion $J[3]$. The equations defining $J[3]$ as a scheme are well-known in genus 2 (see §4.1 or [9]) and we use Gröbner basis machinery to convert them essentially to a univariate equation of degree $80 = |J[3] \setminus \{0\}|$. The problem then becomes to compute the Galois group of this polynomial, and enough information about the inertia action on the roots to reconstruct the conductor. This is the core of the paper (§4). In particular, we discuss how to guarantee that the results are provably correct (§4.3).

As for the tame part, it can be computed from the regular model of C/K, which is in principle accessible: take any model of C over the ring of integers of K, and perform repeated blowups until it becomes regular[1]. However, the algorithm to compute a regular model is currently only partially implemented in Magma [8], and so we complement our algorithm with a result that determines n_{tame} from elementary invariants, in the majority of the cases (Theorem 3.1).

An alternative approach to getting the conductor would be to find a Galois extension F/K where C acquires semistable reduction and a semistable model

[1]Then $n_{\text{tame}} = 4 - 2d_a - d_t$, where d_a ('abelian part') is the sum of genera of reduced components of the special fibre of the model, and d_t ('toric part') is the number of loops

over F, and analyse the action of inertia of F/K on the model. From this one can determine the l-adic representation $V_l J$, in particular the conductor exponent; see e.g. [16, §6]. Moreover, that there are more compact polynomials defining such an F in the case of genus 2, $p = 2$ than the degree 80 3-torsion polynomial. For example, there is the monodromy polynomial of Lehr-Matignon in the potentially good reduction case, of degree 16 [42, §3]. However, the splitting field of any such polynomial would have ramification degree no less that that of $K(J[3])/K$, by the Serre-Tate theorem [61, Cor. 2]. So such a field (and the model of C over it) would be still prohibitively large to compute, and our algorithm avoids this.

Regarding Gröbner bases, the algorithm would be accelerated by an algorithm to solve multivariate systems of equations p-adically (see Remark 5.1). This is also work in progress. Finally, it should be possible to extend the algorithm to compute the conductor to function fields of characteristic 2 as well, by modifying the equations of the curve and its 3-torsion in §4.1 appropriately.

This algorithm has been implemented as a Magma package [24], and has been used to verify all of the genus 2 curves in the LMFDB (§6).

2 Notation

Throughout the paper, we use the following notation:

$K, L, ...$	local fields, of residue characteristic p
$\mathcal{K}, \mathcal{L}, ...$	global fields
G_K	$= \mathrm{Gal}(\bar{K}/K)$, the absolute Galois group of K
$I_K < G_K$	its inertia group
T	$\mathbb{Z}_l$-module with an action of G_K, with $l \neq p$
V	the associated l-adic representation $T \otimes_{\mathbb{Z}_l} \mathbb{Q}_l$
$\bar{V}$	the reduction $T \otimes_{\mathbb{Z}_l} \mathbb{F}_l$
G^u	upper numbering of ramification groups
G_v	lower numbering of ramification groups
$n = n_{\mathrm{tame}} + n_{\mathrm{wild}}$	conductor exponent

We are interested in the situation that J/K is an abelian variety, $T = T_l J$ is its l-adic Tate module, $V = V_l J$ and $\bar{V} = J[l]$ is its l-torsion. Recall that the

conductor exponent of such a representation is given by (see e.g. [67])

$$n(V) = \int_{-1}^{\infty} \operatorname{codim} V^{G_K^u} du,$$

with

$$n_{\text{tame}}(V) = \int_{-1}^{0} \qquad \text{and} \qquad n_{\text{wild}}(V) = \int_{0}^{\infty}.$$

For $u > 0$, G_K^u is pro-p, and [67, §6]

$$\operatorname{codim} V^{G_K^u} = \operatorname{codim} \bar{V}^{G_K^u}.$$

Our approach is that we will compute $n_{\text{tame}}(V)$ as the codimension of inertia invariants V^{I_K}, and the wild conductor exponent as

$$n_{\text{wild}}(V) = \int_{0}^{\infty} \operatorname{codim} J[l]^{G_K^u} du,$$

and replacing G_K by $\operatorname{Gal}(K(J[l])/K)$.

3 Tame conductor exponent

Let K be any non-Archimedean local field, J/K a g-dimensional abelian variety, and l a prime different from the residue characteristic of K. Write $T = T_l J$ for the l-adic Tate module of J/K and $V = V_l J = T_l J \otimes_{\mathbb{Z}_l} \mathbb{Q}_l$, both viewed as representations of the inertia group $I_K < G_K$.

Recall[2] that there is a canonical filtration on T coming from the toric part and the abelian part of J over a field where it acquires semistable reduction. With respect to this filtration, I_K acts on T as

$$\begin{pmatrix} \chi & * & N \\ 0 & \rho & * \\ 0 & 0 & \hat{\chi} \end{pmatrix} \tag{3.1}$$

[2]These are 'standard' facts that we found a little hard to locate in the literature, but they are summarised in [11] §2.10: for the existence of a $\operatorname{Gal}(\bar{K}/K)$-stable filtration that forces the Galois group action to be upper-triangular see [11, p.13, 2nd half]; for the fact that the representations on the graded pieces χ and ρ are independent of l see [11, p.13, bottom], and for the maps between them and the monodromy pairing [11, pp. 12,14]. See also forthcoming paper [17].

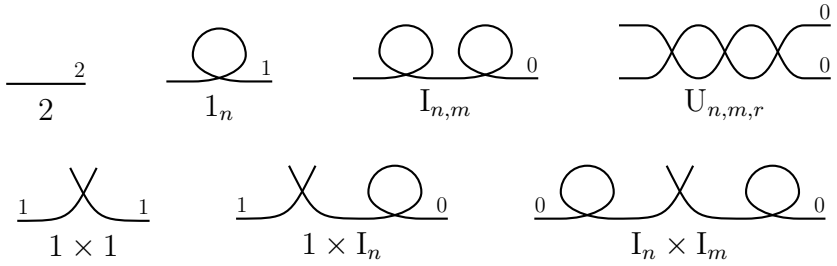

Figure 1: The 7 stable reduction types for genus 2.

with $\chi : I_K \to \mathrm{GL}_t(\mathbb{Z}_l)$, $\rho : I_K \to \mathrm{GL}_{2a}(\mathbb{Z}_l)$ continuous with finite image (t='toric', a='abelian', $2t+2a = \mathrm{rk}_{\mathbb{Z}_l} T = 2g$), and $\hat{\chi}$ the dual of χ. The 'monodromy matrix' N has $\mathbb{Z}$-coefficients, and χ factors through $\mathrm{GL}_t(\mathbb{Z})$ as well. In particular, $\chi \otimes \mathbb{Q}_l$ is self-dual with determinant of order 1 or 2. Consequently, the same holds for $\rho \otimes \mathbb{Q}_l$, as $\det(3.1) = 1$ by the Weil pairing.

Now, we specialise to the case when $J = \mathrm{Jac}\, C$ is the Jacobian of a genus 2 curve and $l = 3$. We will explain in §4 how to compute the image I of I_K in $\mathrm{Aut}\, J[3]$ and the dimension of inertia invariants $\dim J[3]^I$.

We can also compute t and a using a theorem of Liu [43, Thm 1] that determines the stable type of C/K from the Igusa invariants of the curve. There are 7 possible stable types in genus 2, in other words possibilities for stable reduction. (For elliptic curves there are 2 types of stable reduction — good and multiplicative.) They are listed as cases I, II, ..., VII in Liu's theorem, and in the notation of [20] they are denoted 2, 1_n, $\mathrm{I}_{n,m}$, $\mathrm{U}_{n,m,r}$, 1×1, $1 \times \mathrm{I}_n$, $\mathrm{I}_n \times \mathrm{I}_m$. The special fibres are shown in Figure 1, with numbers above the components indicating geometric genus.

Of these, types 2 and 1×1 have $t = 0, a = 2$ (potentially good reduction of J), types 1_n and $1 \times \mathrm{I}_n$ have $t = a = 1$ (mixed), and $\mathrm{I}_{n,m}$, $\mathrm{U}_{n,m,r}$ and $\mathrm{I}_n \times \mathrm{I}_m$ have $t = 2, a = 0$ (potentially totally toric reduction).

The main result of this section recovers the tame conductor exponent of J/K from the invariants I, $\dim J[3]^I$ and t, when this is possible:

Theorem 3.1. *Let K be a non-Archimedean local field of residue characteristic $\neq 3$ and C/K a genus 2 curve with Jacobian J/K. Write*

I = *image of inertia* $I_K < G_K$ *in* $\mathrm{Aut}\, J[3]$ *(so* $I < \mathrm{Sp}_4(\mathbb{F}_3)$*),*
d = $\dim(V_3 J)^I$ *(so* $0 \le d \le 4$*),*
$\bar{d}$ = $\dim J[3]^I$ *(so* $0 \le \bar{d} \le 4$*),*
t = *potential toric dimension of* J *(so* $0 \le t \le 2$*),*
f = $4 - d = n_{tame}(V_3 J) = n_{tame}(J/K)$ *(so* $0 \le f \le 4$*).*

Then $\bar{d} \ge d$ *and so* $f \ge 4 - \bar{d}$*. Moreover,*

1. *If* $\bar{d} = 0$ *then* $f = 4$*.*

2. *If* $\bar{d} = 4$ *then* $d = 4 - t$ *and* $f = t$*.*

3. *Suppose* J *has potentially good reduction (*$t = 0$*). If* $|I| = 3$ *and* $\bar{d} = 2$ *then* $f = 4$*; in all other cases,* f *is the smallest even integer* $\ge 4 - \bar{d}$*.*

4. *If* $(t, |I|) \in \{(1,3), (2,3), (1,2), (1,6)\}$ *then* f *is not uniquely determined as a function of* t*,* I *and* $\bar{d}$*.*

5. *If* $(t, |I|) = (2, 9)$ *then* $f = 4$*; in all other cases not covered,* $f = 3$*.*

Proof. Write $T = T_3 J$, $V = V_3 J$. Note that after tensoring (3.1) with $\mathbb{Q}_3$ and a suitable change of basis, both $*$'s can be made 0 and N a $t \times t$ identity matrix. In particular,

$$V^{I_K} = \chi^I \oplus \rho^I, \qquad f = 4 - \dim \chi^I - \dim \rho^I. \tag{3.2}$$

If V has an I_K-invariant subspace of dimension d, its intersection with T gives a rank d saturated sublattice of T, whose reduction contributes at least dimension d to $J[3]^I$. This shows that $\bar{d} \ge d$, and implies (1).

(2) By Raynaud's semistability criterion [33, Prop 4.7], J is semistable if $J[m]$ is unramified for some $m \ge 3$ coprime to the residue characteristic. Here I_K acts trivially on $J[3]$, and so J is semistable. In other words, $f = t$ and $d = 4 - t$.

For the remainder of the proof, we assume $\bar{d} \in \{1, 2, 3\}$.

(3) By Serre-Tate's theorem [61, Cor. 2], J has good reduction over $K(J[3])$; that is, I_K acts on $V_3 J$ through I. By Poincare duality, this representation has even-dimensional inertia invariants, in other words d is even. As $d \le \bar{d} \in \{1, 2, 3\}$, the only possibility for $f = 4 - d$ not to be the smallest even integer $\ge 4 - \bar{d}$ is when $d = 0$ and $\bar{d} \in \{2, 3\}$. Suppose we are in that case.

Consider the possibilities for $I < \mathrm{Sp}_4(\mathbb{F}_3)$. Note that 3 divides $|I|$, for otherwise the classical representation theory of I agrees with its modular representation over $\mathbb{F}_3$, implying $d = \bar{d}$. Also note that $C_3 \times C_3$ is not a quotient of I, as the residue characteristic of K is not 3, and tame inertia is cyclic. Computing in Magma [8], we find that that $\mathrm{Sp}_4(\mathbb{F}_3)$ has 162 conjugacy classes of subgroups, of which 5 satify the three properties (a) order multiple of 3, (b) no $C_3 \times C_3$-quotient, and (c) $\bar{d} \in \{2, 3\}$. Call them $H_1, H_2, H_3 \cong C_3$, $H_4 \cong C_6$ and $H_5 \cong \mathrm{SL}_2(\mathbb{F}_3)$.

By the classification of integral C_p-lattices [13, 54], there are two indecomposable $\mathbb{Z}_3[C_3]$-lattices, up to isomorphism: the trivial lattice of rank 1, and a lattice Λ of rank 2 on which the generator of C_3 acts as $\left(\begin{smallmatrix} -1 & 1 \\ -1 & 0 \end{smallmatrix}\right)$; every finite rank $\mathbb{Z}_3[C_3]$-lattice is a direct sum of these. If $I \cong C_3$, then as $d = 0$, we must have $T \cong \Lambda \oplus \Lambda$, and it has $\bar{d} = 2$ as claimed.

It remains to show that $I \in \{H_4, H_5\}$ with $d = 0$ is impossible. Suppose we are in this case, and let $z \in I$ be the unique central element of order 2. As above, the classical representation theory of the group $\langle z \rangle \cong C_2$ agrees with its modular representation over $\mathbb{F}_3$. In both H_4 and H_5 the action of z on $\bar{V} = J[3]$ has two $+1$ and two -1 eigenvalues. The same is therefore true for V; moreover, $V = V^+ \oplus V^-$ and $T = T^+ \oplus T^-$ decompose into the two 2-dimensional eigenspaces for z and this decomposition induces the one on $J[3]$.

The group $\mathrm{SL}_2(\mathbb{F}_3)$ has 3 one-dimensional complex representations factoring through $\mathrm{SL}_2(\mathbb{F}_3)/Q_8 \cong C_3$, three faithful 2-dimensional ones in which z acts as -1, and a 3-dimensional one with z acting as $+1$. Thus, when I is H_4 and H_5, the space T^+ must be a representation of the unique C_3 quotient of I. It has no trivial subrepresentations (as $d = 0$), so $T^+ \cong \Lambda$ as a $\mathbb{Z}_3[C_3]$-module. But then

$$\bar{d} = \dim(\Lambda \otimes \mathbb{F}_3)^{C_3} + \dim(T^- \otimes \mathbb{F}_3)^I = 1 + 0,$$

contradicting the assumption $\bar{d} \in \{2, 3\}$.

(4) The following curves give examples over $\mathbb{Q}_2$ that prove that f is not a function of t, I and $\bar{d}$, as claimed. (In each case, f can be determined by computing

the regular model.)

t	I	$\bar{d}$	f	$C/\mathbb{Q}_2$
1	C_3	3	1	$y^2 = x^6 + 4x^4 + 2x^3 + 4x^2 + 1$
1	C_3	3	3	$y^2 = 4x^6 - 20x^4 - 8x^3 + 21x^2 + 22x + 13$
2	C_3	2	2	$y^2 = x^6 + 6x^4 - 7x^2 + 16$
2	C_3	2	4	$y^2 = 5x^6 + 4x^3 - 12$
1	C_2	2	2	$y^2 = -x^6 + 6x^4 - x^2 - 8$
1	C_2	2	3	$y^2 = x^6 - 6x^4 + x^2 + 8$
1	C_6	1	3	$y^2 = x^6 - 6x^4 + 5x^2 + 8$
1	C_6	1	4	$y^2 = x^6 - 31x^4 - 25x^2 - 32$

(5) To deal with all the remaining cases, first suppose that J has totally toric reduction over $K(J[3])$, in other words $t = 2$. In the notation of (3.1), we have a homomorphism

$$\chi : I \longrightarrow \mathrm{GL}_2(\mathbb{Z}) \quad (\hookrightarrow \mathrm{GL}_2(\mathbb{Z}_3))$$

whose image we denote by $\bar{I}$ and whose kernel is C_1 or C_3. Finite subgroups of $\mathrm{GL}_2(\mathbb{Z})$ are contained in D_4 or D_6. Of those, D_3, D_6 only occur as inertia groups in residue characteristic 3, and C_2^2, C_4, C_6, D_4 have an element acting as -1, forcing $\bar{d} = 0$ (case (1)).

The remaining possibilities are

$$\bar{I} \in \{C_1, C_2, C_3\}, \qquad I \in \{C_1, C_2, C_3, C_6, C_9\}.$$

We have excluded $I = C_1$ (case (1)) and $I = C_3$ (case (4)). When $I = C_9$, its image $\bar{I} \cong C_3$ has no invariants, and so $f = 4$ (proving the case $(t, |I|) = (2, 9)$).

The only remaining case is $\bar{I} = C_2$, acting with eigenvalues $+1, -1$ (otherwise $\bar{d} \in \{0, 4\}$ again). In this case, the full action on T is of the form

$$\begin{pmatrix} 1 & 0 & * & 0 \\ 0 & -1 & 0 & * \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

in some basis, with non-zero $*$'s. This has one-dimensional invariants, and so $f = 3$, as claimed.

Finally suppose $t = 1$, so that I_K acts on T as

$$\begin{pmatrix} \chi & * & * & \neq 0 \\ 0 & a & b & * \\ 0 & c & d & * \\ 0 & 0 & 0 & \chi \end{pmatrix}$$

As before, write ρ for the representation $I_K \to \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$. Because I is not one of the already excluded groups C_1, C_2, C_3, C_6, the image of I_K under $\bar{\rho} = \rho \mod 3$ is not C_1 or C_2. But any other subgroup of $\mathrm{GL}_2(\mathbb{Z}_3)$ of finite order is either D_3, which cannot be a local Galois group, or $\bar{\rho}(I)$ has no invariants on $\mathbb{F}_3^2$. Hence $\bar{\rho}^{I_K} = 0$, and $J[3]^{I_K} = \chi^{I_K}$ has either dimension 0 (case (1)) or dimension 1 with $f = 3$, as claimed. □

4 Wild conductor exponent

Recall that we wish to compute

$$n_{\text{wild}} = \int_0^\infty \operatorname{codim} J[3]^{G^u} du$$

where $G = G_K$. Note, however, that G_K acts on $J[3]$ through its finite quotient $\mathrm{Gal}(K(J[3])/K)$ so we may equally well take $G = \mathrm{Gal}(K(J[3])/K)$ or any quotient in between.

The integrand here is decreasing, non-negative, integral and left-constant, so if we denote by $u_1 = 0, u_2, \ldots, u_t$ the jump points in the integrand, then we get

$$n_{\text{wild}} = \sum_{i=2}^{t} (u_i - u_{i-1}) \operatorname{codim} J[3]^{G^{u_i}}.$$

Let $Z \in J[3]$ be a 3-torsion point and let $L = K(Z)$ be the extension it generates. Then Z is fixed by G^u if and only if L is fixed by G^u. Since $G^u \triangleleft G$, this occurs if and only if any K-conjugate of Z is fixed by G^u. If

$\hat{u} = \hat{u}(L/K) = \inf\{u : L \text{ fixed by } G^u\}$ denotes the highest upper ramification break of L/K, then this occurs if and only if $\hat{u} \leq u$.

Hence, if $Z_1, \ldots, Z_m$ are representatives of the K-conjugacy classes of $J[3]$, generating extensions L_i/K with highest upper ramification break $\hat{u}_i$ then letting $u_0 = -1 < u_1 = 0 < \ldots < u_t$ be the sorted elements of $\{-1, 0, \hat{u}_1, \ldots, \hat{u}_m\}$ we deduce

$$n_{\text{wild}} = \sum_{i=2}^{t} (u_i - u_{i-1}) \left(2g - \log_3 \sum_{j:\hat{u}_j \leq u_i} (L_j : K) \right)$$

since $2g = \dim V$ and $(L_j : K)$ is the number of K-conjugates of Z_j.

We proceed by finding the extensions L_i/K explicitly, from which we compute n_{wild} via this equation.

4.1 Equation for 3-torsion of genus 2 curves

As before, let C/K be a curve of genus 2, with Jacobian J. The linear system for the canonical divisor on C yields a standard model

$$C : y^2 = f(x), \qquad \deg f = 5 \text{ or } 6.$$

The following statement is well-known (see e.g. [9] proof of Lemma 3); in fact, it works over any field of characteristic $\neq 2, 3$.

Proposition 4.1. *Non-zero elements of $J[3]$ are in 1-1 correspondence with ways of expressing f in the form*

$$f = (z_4x^3 + z_3x^2 + z_2x + z_1)^2 - z_7(x^2 + z_6x + z_5)^3, \qquad z_i \in \bar{K}, \qquad (*)$$

and this correspondence preserves the action of G_K.

Explicitly, suppose D is a divisor on C,

$$D = (P_1) + (P_2) - (\infty_1) - (\infty_2), \qquad P_i = (X_i, Y_i)$$

for which $3D$ is principal, say $3D = \operatorname{div} g$. Then $g \in \langle 1, x, x^2, x^3, y \rangle$. After a

(unique) re-scaling, say

$$g = y + b_3x^3 + b_2x^2 + b_1x + b_0.$$

The norm

$$\begin{aligned}\mathrm{Norm}_{K(C)/K(x)}(g) &= (b_3x^3+b_2x^2+b_1x+b_0-y)(b_3x^3+b_2x^2+b_1x+b_0+y)\\ &= (b_3x^3+b_2x^2+b_1x+b_0)^2 - f.\end{aligned}$$

is a function on $\mathbb{P}^1$ whose divisor $3(X_1)+3(X_2)-6(\infty)$ is a cube, and so

$$(b_3x^3 + b_2x^2 + b_1x + b_0)^2 - f = c_2(x^2 + c_1x + c_0)^3,$$

as stated. In this form,

$$X_{1,2} = \text{ roots of } x^2+c_1x+c_0 = 0, \qquad Y_i = -b_3X_i^3-b_2X_i^2-b_1X_i-b_0.$$

We view $(*)$ as giving a system of 7 equations in the 7 variables z_i.

4.2 Finding the 3-torsion fields

Our goal, then, is to find the (K-isomorphism classes of) fields L/K generated by the (K-conjugacy classes of) solutions Z to the system of equations $(*)$.

A general tool used to solve systems of polynomial equations such as this is to compute a Gröbner basis for the polynomial ideal generated by the polynomials. Generically, a reduced sorted minimal Gröbner basis with respect to the lexicographic ordering on variables will be a finite sequence of polynomials such that the first is univariate, the second is a polynomial in two variables, and so on. Then to solve the system, we first find a root of the first polynomial; then we substitute this value into the second polynomial, yielding a polynomial in one variable, and we find a root of this; we repeat this procedure. In the end, this will produce a sequence of roots which together are a solution to the system.

For our system in particular, the 80 roots come in pairs of the form

$$(Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7), \quad (-Z_1, -Z_2, -Z_3, -Z_4, Z_5, Z_6, Z_7),$$

and so generically there are 40 distinct values for Z_7, for each of these there is a unique value for Z_6 and Z_5 and two distinct values for Z_4, and for each of these there is a unique value for Z_3, Z_2 and Z_1.

In this generic case, the Gröbner basis described above will be a sequence of 7 polynomials $B_1, \ldots, B_7 \in K[z_1, \ldots, z_7]$ such that $B_i \in K[z_i, \ldots, z_7]$, $\deg_{z_i} B_i = d_i$ where $d = (1, 1, 1, 2, 1, 1, 40)$.

Following the above discussion on solving systems using Gröbner bases, we first factorize $B_7 \in K[z_7]$ (of degree 40), let g be one of its irreducible factors, let M/K be the extension it defines, and let $Z_7 \in M$ be a root of g. Substituting this into $B_6 \in K[z_6, z_7]$ we get $B_6(z_6, Z_7) \in M[z_6]$, which is linear, and let Z_6 be its root. Similarly we let Z_5 be the root of $B_5(z_5, Z_6, Z_7) \in M[z_5]$. Next, $B_4(z_4, Z_5, Z_6, Z_7) \in M[z_4]$ is quadratic, so we factorize it, let h be one of its factors, let L/M be the extension it defines, and let $Z_4 \in L$ be a root of h. Continuing, we find unique Z_3, Z_2 and Z_1 which together produces a solution $Z = (Z_1, \ldots, Z_7)$. Repeating this for all factors g and h we find all solutions Z of the system (up to conjugacy) and the extensions L/K which they define.

If we are not in this generic case, then the Groeber basis is not of this form and there is some coincidence in the coordinates of some solutions of the 7 equations. If we apply a random Möbius transformation $x \mapsto \frac{ax+b}{cx+d}$ to the defining polynomial $f(x)$ then the curve it defines is isomorphic to the original but the solutions Z have moved, probably to the generic case. In practice, a small number of Möbius transformations is ever necessary to put the solutions into the generic case.

Remark 4.2. An algorithm of this sort would work with any ordering on $\{z_1, \ldots, z_7\}$. This ordering was chosen because it allows us to factor a degree-40 polynomial followed by a quadratic, which is somewhat faster than just factoring a degree-80 polynomial required for other orderings.

4.3 Provability

In practice, however, computing a Gröbner basis of this sort is difficult. Gröbner basis algorithms require exact fields, so in practice we represent K as a completion of a number field $\mathcal{K}$ at some place $\mathfrak{p} \mid 2$, and $f(x) \in \mathcal{K}[x]$.

The best known algorithm over number fields (and indeed the only algorithm

which appears to run in feasible time on our problem) computes the basis modulo many primes and finds the global basis via the Chinese remainder theorem. The problem here is that a priori we cannot determine the size of the coefficients, and so a heuristic is used to decide if we have used enough primes to get the answer. The result is that the algorithm does not yield provable results. Nevertheless, it is possible to prove the output of the previous algorithm as follows.

If the Gröbner basis algorithm was correct, then any $Z = (Z_1, \ldots, Z_7)$ should be a solution to the original system of 7 equations $(*)$ over K. With the following version of Hensel's lemma, we can show that Z is indeed very close to a unique genuine solution, and we can say how close.

The following version of Hensel's lemma is standard (see e.g. [41] Thm. 23 with $t = \det J_f(b)$, $s = vf(b)$ and $vJ_f^*(b)f(b) \geq s$).

Theorem 4.3 (Hensel's lemma for multivariate systems)**.** *Suppose K is a local field and $F = (F_1, \ldots, F_m) \in \mathcal{O}_K[z_1, \ldots, z_m]$ is a system of m equations in m variables over $\mathcal{O}_K$ and $Z = (Z_1, \ldots, Z_m) \in \mathcal{O}_K^m$. Let $s = \min_i v_K(F_i(Z))$ and let $t = v_K J(F)(Z)$ where $J(F)$ denotes the Jacobian determinant of F (the determinant of the $m \times m$ matrix whose (i,j)th entry is $\frac{\partial F_i}{\partial z_j}$). If $s > 2t$ then there is a unique $Z' \in \mathcal{O}_K^m$ such that $F(Z') = 0$ and $\min_i v_K(Z_i' - Z_i) \geq s - t$.*

Since evaluating resultants, Jacobians and polynomials are just basic arithmetic, these operations can be performed provably, and hence applying Hensel's lemma we prove that each Z is indeed close to a unique solution Z' of the system of equations. Furthermore, Hensel's lemma gives us a method to compute Z' to any prescribed precision. We expect that $Z = Z'$ but we do not prove so.

It remains to check that these solutions Z' generate the fields L and that they are distinct up to K-conjugacy.

Recall that we have $L/M/K$ with $M = K(Z_7)$, $g(x) \in K[x]$ the minimal polynomial for Z_7, and $L = M(Z_4)$, $h(x) \in M[x]$ the minimal polynomial for Z_4. We also have $Z_7', Z_4' \in L$ and want to prove that $L = K(Z_7', Z_4')$. Since we expect that $Z_7' = Z_7$, then we expect Z_7' is closer to Z_7 than any other root of g, and so by Krasner's lemma we conclude that $M = K(Z_7) \subset K(Z_7')$. Another application of Krasner's lemma on h and Z_4' implies that $L = M(Z_4) \subset M(Z_4')$.

Combining these, we deduce $L = M(Z_4) \subset M(Z_4') \subset K(Z_4', Z_7') \subset L$ and hence $L = K(Z_4', Z_7') = K(Z')$.

To check Krasner's lemma on a polynomial $h \in K[x]$ and some $Z \in \bar{K}$, note that it is equivalent to check that there is a root of $h(x+Z)$ of higer valuation than all others. It is well-known that the Newton polygon of a polynomial measures the valuations of its roots, and therefore Krasner's lemma is applicable if and only if the Newton polygon of $h(x + Z)$ has a vertex with abscissa 1. This condition is explicitly checkable.

Finally, if Z_7 is a root of a factor g of B_7 and Y_7 is a root of a different factor of B_7, then $g(Z_7) = 0 \neq g(Y_7)$, so if we check that $v(g(Z_7')) > v(g(Y_7'))$ then we have proven that $Z_7' \neq Y_7'$. Performing a similar check on pairs of Z_4' determines that they are different. Together, this will prove that each pair of solutions is distinct.

By performing all these checks with large enough precision, we can determine whether or not the Z are a genuine set of distinct solutions generating the right fields. If any of these checks fails, then the Gröbner basis algorithm was incorrect, and we should try the algorithm again with a lower heuristic chance of failure.

Remark 4.4. There is a conceptually simpler method for provability. Letting $I \lhd \mathcal{K}[z_1, \ldots, z_7]$ be the ideal generated by the original system $(*)$, and letting J be the ideal generated by the Gröbner basis, then we wish to prove that $I = J$. Since J is generated by a Gröbner basis, there is a normal form for reduction modulo J and hence we can check that each generator of I is zero mod J and so deduce $I \lhd J$. Additionally we know a priori that I has precisely 80 solutions, and from the structure of the Gröbner basis that J has precisely 80 solutions. Combined, this implies $I = J$.

We call this the **global proof method** to distinguish it from the **local proof method** above. In practice, unless the coefficients of $f(x)$ are very small, the global method takes much longer than the local method. Over $\mathbb{Q}$, with small coefficients, the global method is typically around twice as quick, but this benefit quickly diminishes as the field degree increases.

4.4 Tame conductor exponent revisited

In order to compute the tame conductor exponent using Theorem 3.1, we require $\bar{d} = \dim J[3]^{G_0}$ and $|I|$. In previous sections we have already seen an algorithm to compute $\dim J[3]^{G^u}$ for any u having already computed L_j/K, so this is easy as a side-effect of previous work.

For $|I|$, consider $e = \operatorname{lcm}_j e(L_j/K)$, which again is easy to compute from L_j/K. Clearly it is a divisor of $|I|$. The following lemma shows that e is a good enough guess at $|I|$ in the sense that the statement of Theorem 3.1 depends only on t, e and $\bar{d}$.

Lemma 4.5. *Let* $S = \{1, 2, 3, 4, 5, 6, 9, 10, 12, 18\}$. *If* $e \in S$ *or* $|I| \in S$ *then* $|I| = e$. *If* $e = 80$ *then* $|I| = 160$. *If* $e \in \{8, 24\}$ *then* $|I| \in \{8, 24\}$. *Otherwise* $e \in \{16, 32, 48, 64\}$ *and* $|I| \in \{16, 32, 48, 64, 96, 128, 192, 384\}$.

Proof. Properties of the Weil pairing imply that $I < \mathrm{Sp}_4(\mathbb{F}_3)$. Letting W be the 2-Sylow subgroup of I, ramification theory implies $W \triangleleft I$ and I/W cyclic. The lemma is proven by checking all groups I consistent with these facts. □

5 The algorithm

We use the following algorithm to compute the highest upper ramification break $\hat{u}(L/K)$. It takes as input the extension L/K and returns the sequence $(u_i, v_i, s_i)_{i=0}^t$ where $v_0 = -1 < v_1 < \ldots < v_t$ are the breaks in the ramification filtration of L/K in the lower numbering, u_i are the corresponding breaks in the upper numbering, and $s_i = |\Gamma_{v_i}|$ are the sizes of the corresponding ramification subsets of the Galois set Γ of K-embeddings $L \to \bar{K}$. In particular, $\hat{u}(L/K) = u_t$.

See Chapter III for the definition of the ramification polynomial (the coefficients of which have valuation r_i in the algorithm), the ramification polygon P and its connection to the ramification filtration of L/K. See Chapter IV, §9.6 for the connection of this filtration to the upper and lower ramification breaks and the Galois set Γ.

1: *(Compute the ramification polygon of L/U)*
2: $U \leftarrow$ the maximal unramified subextension of L/K
3: $e \leftarrow (L : U)$
4: $E \leftarrow$ a defining Eisenstein polynomial for L/U
5: $r_i \leftarrow \min_{j=i}^{e-1} v(E_j\binom{j}{i}) + \frac{j}{e}$ for $i = 1, \ldots, e$
6: $P \leftarrow$ the lower convex hull of the points (i, r_i) for $1 \leq i \leq e$

7: *(Compute u_i, v_i and $s_i = |\Gamma_{v_i}|$)*
8: $u_0 \leftarrow -1$
9: $v_0 \leftarrow -1$
10: $s_0 \leftarrow (L : K)$
11: $t \leftarrow$ the number of faces of P
12: **for all** $i = 1, \ldots, t$ **do**
13: $\quad F \leftarrow$ the ith face of P from the right
14: $\quad v_i \leftarrow$ the negative of the gradient of F
15: $\quad s_i \leftarrow$ the abscissa of the right hand vertex of F
16: $\quad u_i \leftarrow u_{i-1} + \frac{s_{i-1}}{s_0}(v_i - v_{i-1})$
17: **end for**
18: **return** $((u_i, v_i, s_i))_{i=0}^{t}$

Now we present the final algorithm, which takes a polynomial $f(x) \in \mathcal{K}[x]$ of degree 5 or 6 over a number field $\mathcal{K}$ defining a hyperelliptic curve $y^2 = f(x)$, and a prime ideal $\mathfrak{p}$ of $\mathcal{K}$ dividing 2, and returns the conductor exponent $n_{\mathfrak{p}}$ of the curve at $\mathfrak{p}$.

1: *(Apply Möbius transformations to $f(x)$ until its 3-torsion points are in general position)*
2: **repeat**
3: $\quad$ choose $a, b, c, d \in \mathbb{Z}$ so that $ad - bc \neq 0$
4: $\quad \tilde{f} \leftarrow f(\frac{ax+b}{cx+d})(cx + d)^6$
5: $\quad F = (F_i)_{i=1}^{7} \leftarrow$ coefficients of

$$(z_1 + z_2x + z_3x^2 + z_4x^3)^2 + z_7(z_5 + z_6x + x^2)^3 - \tilde{f}(x)$$

6: $\quad B = (B_i)_i \leftarrow$ Gröbner basis of F

7: **until** B is in generic form

8: *(Find the fields defined by each Z_7)*
9: $K \leftarrow \mathcal{K}_\mathfrak{p}$
10: $S \leftarrow$ empty sequence
11: $C \leftarrow$ empty sequence
12: $(g_i)_i \leftarrow$ irreducible factorization of $B_7(x)$ over K
13: **for all** g_i **do**
14: $M \leftarrow$ the extension of K defined by g_i
15: $Z_7 \leftarrow$ a root of g_i in M
16: $Z_6 \leftarrow$ the root of linear $B_6(x, Z_7)$ over M
17: $Z_5 \leftarrow$ the root of linear $B_5(x, Z_6, Z_7)$ over M

18: *(Find the fields defined by each Z_4)*
19: $(h_i)_i \leftarrow$ irreducible factorization of $B_4(x, Z_5, Z_6, Z_7)$ over M
20: **for all** h_i **do**
21: $L \leftarrow$ the extension of M defined by h_i
22: $Z_4 \leftarrow$ a root of h_i in L
23: $Z_3 \leftarrow$ the root of linear $B_3(x, Z_4, Z_5, Z_6, Z_7)$ over L
24: $Z_2 \leftarrow$ the root of linear $B_2(x, Z_3, Z_4, Z_5, Z_6, Z_7)$ over L
25: $Z_1 \leftarrow$ the root of linear $B_1(x, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7)$ over L

26: *(Check the solutions are valid with Hensel's lemma)*
27: **assert** Z is Hensel liftable to a solution of F
28: $Z' \leftarrow$ the Hensel-lifted solution (we expect $Z' = Z$)

29: *(Check the solutions generate the right fields with Krasner's lemma)*
30: **assert** the Newton polygon of $g_i(x + Z'_7)$ has a vertex above 1
31: **assert** the Newton polygon of $h_j(x + Z'_4)$ has a vertex above 1

32: *(Check the solutions are distinct)*
33: **for** $(Y'_7, Y'_4) \in C$ **do**
34: **assert** $v_K(g_i(Z'_7)) > v_K(g_i(Y'_7))$ or $v_K(h_i(Z'_4)) > v_K(h_i(Y'_4))$
35: **end for**

append (Z_7', Z_4') to C

(Save L)
append L to S
end for
end for

(Compute the tame and wild exponents from S)
$\bar{d} \leftarrow$ the function $u \mapsto \log_3(1 + \sum_{L \in S : \hat{u}(L/K) \leq u} (L : K))$ $(= \dim \bar{V}^{G^u})$
$e \leftarrow \operatorname{lcm}_{L \in S} e(L/K)$
$t \leftarrow$ potential toric dimension of J
if $\bar{d}(0) = 0$ **then**
$n_{\text{tame}} \leftarrow 4$
else if $\bar{d}(0) = 4$ **then**
$n_{\text{tame}} \leftarrow t$
else if $t = 0$ **then**
if $e = 3$ and $\bar{d}(0) = 2$ **then**
$n_{\text{tame}} \leftarrow 4$
else
$n_{\text{tame}} \leftarrow$ smallest even integer $\geq 4 - \bar{d}(0)$
end if
else if $(t, e) \in \{(1,3), (2,3), (1,2), (1,6)\}$ **then**
$n_{\text{tame}} \leftarrow$ the tame exponent, computed from a regular model
else if $(t, e) = (2, 9)$ **then**
$n_{\text{tame}} \leftarrow 4$
else
$n_{\text{tame}} \leftarrow 3$
end if
$u_0, \ldots, u_t \leftarrow$ the sorted elements of $\{\hat{u}(L/K) : L \in S\} \cup \{-1, 0\}$
$n_{\text{wild}} \leftarrow \sum_{i=2}^{t} (u_i - u_{i-1})(4 - \bar{d}(u_i))$
return $n_{\text{wild}} + n_{\text{tame}}$

Remark 5.1. Note that the approach to solving the system of 7 equations in 7 variables is to compute a Gröbner basis globally, and then solve this system

locally. This is the only global aspect of the algorithm, and becomes the bottleneck when the global coefficients become large. An alternative approach is to solve the system of equations directly locally, perhaps using a Montes-type algorithm similar to univariate factorization algorithms which split the system into several smaller systems. This is the subject of ongoing research (see Chapter VI).

Remark 5.2. Recalling Remark 4.4, if we wish to use the global proof method instead, then we can skip over lines 22–36 and instead insert after line 7 a check that each element of F reduces to 0 modulo B.

6 Implementation

The algorithms described in this paper have been implemented [24] in the Magma computer algebra system [8] using the exact p-adics packages of Chapter IV. The implementation, modulo bugs, produces provable results at every step.

The LMFDB [45] contains the 66,158 genus 2 hyperelliptic curves defined over $\mathbb{Q}$ computed by Booker et al [7]. Of these, all but 1113 have discriminant of 2-valuation less than 12 and therefore their conductor exponent at 2 is computable via Ogg's formula. Our algorithm has been run on the 1113 remaining curves, using the global proof method (see Remark 4.4). The computation took 9.4 core-hours in total on a 2.7GHz Intel Xeon, averaging 30 core-seconds per curve.

For all but 6 of these curves, the fast tame conductor algorithm of §3 succeeds, and so we compute an entire conductor exponent at 2. For 4 of the remaining 6 curves, a regular model was quickly computed by Magma (taking at most 10 seconds) and therefore the tame exponent was deduced this way. For the remaining 2 curves (labelled `3616.b.462848.1` and `18816.d.602112.1` in the LMFDB) a regular model was computed by hand. In all of these cases, the exponent agrees with the unproven results of [7] and therefore we have proven the conductors for all curves in the LMFDB.

The run-time of the algorithm is usually dominated by the factorization of the degree-40 polynomial over K, at least when the defining polynomial $f(x)$ has fairly small coefficients. When these coefficients grow, the (global) Gröbner basis algorithm dominates the run-time.

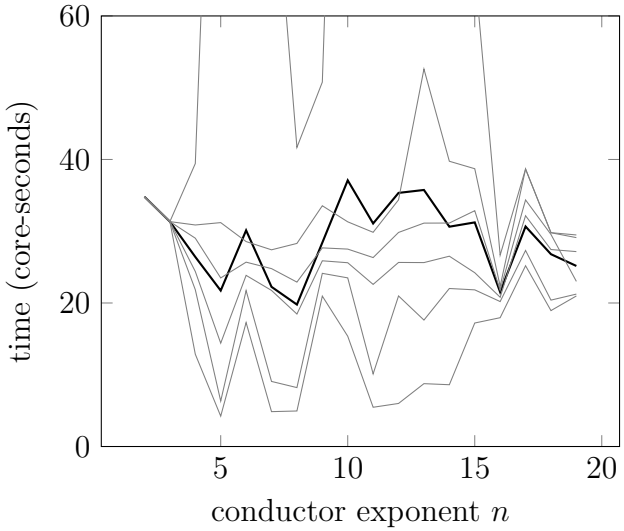

Figure 2: Run-time versus conductor exponent on LMFDB curves. Thick line is mean run-time, thin gray lines are 20-percentiles.

This gives some impetus towards developing a fully local algorithm as suggested in Remark 5.1, since this will be independent of global coefficient sizes.

The implementation has also been tested on some curves defined over quadratic number fields. These results were confirmed by Schembri [59] by finding a corresponding Bianchi modular form whose level squared equals the conductor and proving the expected relationship between their L-functions using Faltings-Serre.

The run-time does not appear to grow much with the conductor exponent, as evidenced by Figure 2 summarizing the run-times of the algorithm on the LMFDB curves.

Chapter VI

Solving Multivariate Systems

1 Introduction

In this article we consider the following problem: given a system $F \in K[x_1, \ldots, x_n]^n$ of n polynomials in n variables over a p-adic field K, what are its roots in K^n?

Note that generically we expect such a system to have finitely many roots, although it could have infinitely many [6, 55]. This article presents an algorithm which will find all roots of a system assuming it has finitely many and that they are all distinct.

Our algorithm is essentially an "OM algorithm" for factoring univariate polynomials over p-adic fields (see e.g. [62, Ch. VI]), but specialised to root-finding and then generalized to multivariate polynomials. We expect that our algorithm can also be generalized to factoring multivariate systems, or more generally still to decomposing p-adic schemes of arbitrary dimension. In this vein, Dokchitser has considered the case of a single equation in two variables [15].

Although presented in terms of p-adic fields, we expect that this whole article generalizes directly to any Henselian discrete valuation field K, with the restriction that Algorithm 6.1 requires the residue class field to be finite (cf. Remark 6.4).

The following definition is important.

Definition 1.1. In this article, a **root** of F is a vector $r \in (\bar{K}^\times)^n$ such that $F(r) = 0$. A **polynomial** is a Laurent polynomial; that is, we allow negative

exponents.

These definitions imply that $v(r) \in \mathbb{Q}^n$ and allow us to freely multiply or divide our polynomials by a monomial without altering its roots, which simplifies things. We expect that some minor modifications will allow us to find all roots allowing zero coordinates.

As a specific point of motivation, consider the algorithm in Chapter V for computing the 2-part of the conductor of a genus-2 hyperelliptic curve. A key part in this algorithm is finding the extensions of $\mathbb{Q}_2$ defined by the 80 roots (corresponding to 3-torsion) of a system of 7 polynomials in 7 variables. The current method is to compute a Gröbner basis over $\mathbb{Q}$ and then use univariate p-adic factoring, but the global step can be expensive. It would be better to factorize the system p-adically directly.

1.1 Layout of article

In §2 we describe Hensel's lemma, which gives the terminating condition of our algorithms.

In §3–§5 we define Newton polytopes and residual systems and give their main properties. These are a generalization of Newton polygons and residual polynomials in the univariate case (see e.g. Chapter IV §9.5 or the related literature on ramification polygons e.g. Chapter III or [32, 53]).

In §6 we give a root-finding algorithm and prove its correctness. It may be slow when the residue class field is large (Remark 6.4).

In §7 we describe an alteration to our earlier algorithm making it more efficient over large residue class fields.

Finally in §8 we give a worked example.

1.2 Notation

We fix a p-adic field K (i.e. a finite extension of $\mathbb{Q}_p$). The ring of integers of K is denoted $\mathcal{O}$, and we fix a uniformizing element π and the valuation v such that $v(\pi) = 1$. The residue class field is $\mathbb{F} = \mathbb{F}_q = \mathcal{O}/(\pi)$ the finite field of order q. If $x \in \mathcal{O}$ then $\bar{x} = x + (\pi) \in \mathbb{F}$ is its residue class. If $x \in K^\times$ then $x^\# = x/\pi^{v(x)} \in \mathcal{O}^\times$.

For vectors $a \in K^n$, we define $v(a) = (v(a_1), \ldots, v(a_n))$ and $\min v(a) = \min_i v(a_i)$. We let $a_{i\ldots j}$ denote the sub-vector $(a_i, \ldots, a_j)$.

2 Hensel's lemma

The following multivariate version of Hensel's lemma is well-known. We include a proof for completeness.

Definition 2.1. If $F \in K[x_1, \ldots, x_n]^n$ then its **Jacobian** is $\mathrm{Jac}(F) = \det(M) \in K[x_1, \ldots, x_n]$ where $M \in K[x_1, \ldots, x_n]^{n \times n}$ is defined pointwise by $M_{i,j} = dF_i/dx_j$.

Lemma 2.2 (Multivariate Hensel's lemma)**.** *Suppose $F \in \mathcal{O}[x_1, \ldots, x_n]^n$ is an integral system of n polynomials in n variables, and $a \in \mathcal{O}^n$. Let $t = v(\mathrm{Jac}(F)(a))$ and suppose there is $s > 2t$ such that $\min v(F(a)) \geq s$. Then there is $b \in \mathcal{O}^n$ such that $\min v(a - b) \geq s - t$, $\min v(F(b)) \geq 2(s - t)$ and $v(J(F)(b)) = t$. In this situation, there is a unique $r \in \mathcal{O}^n$ such that $F(r) = 0$ and $\min v(a - r) \geq s - t$ for all i.*

Proof. Define

$$b = a + \pi^u d$$

for some $d \in \mathcal{O}^n$ and $u \in \mathbb{Z}$ to be determined later. Then

$$F_i(b) \equiv F_i(a) + \pi^u \sum_j F_i^{(j)}(a) d_j \mod \pi^{2u}$$

where $F_i^{(j)}$ denotes the partial derivative of F_i with respect to x_j.

Assume $s \geq u$ and let $c = -F(a)/\pi^u$. Then $c \in \mathcal{O}^n$ and $\min v(c) \geq s - u$. We deduce that $\min v(F(b)) \geq 2u$ if and only if

$$\sum_j F_j^{(j)}(a) d_j \equiv c_i \mod \pi^u.$$

Letting $M_{i,j} = F_i^{(j)}(a)$, then we know that $t = v(\det M) < \infty$ and hence M is invertible. Let $d = M^{-1}c$. Then $\min v(d) \geq \min v(c) - v(\det M) \geq s - u - t$, so letting $u = s - t$ we deduce $d \in \mathcal{O}^n$ as required.

Assuming $s > 2t$ then $s - t > t$, and since $\min v(a - b) \geq s - t > t$, then $v(J(F)(a) - J(F)(b)) > t$, and we deduce $v(J(F)(b)) = t$.

We can now replace a by b and s by $2(s - t) > s$ and repeat. Iterating this process, we get a Cauchy sequence whose limit r is a root of F. □

If we define $a_0 = a$ and $s_0 = s$ and perform this iteration, we find $a_1, a_2, \ldots$ and $s_i = 2(s_{i-1} - t) = 2^i s - (2^i - 1)t$ such that $\min v(a_i - r) \geq s_i - t = 2^i(s - t)$. This procedure is known as **Hensel lifting** and can be used to produce an approximation to r to any desired precision.

The case $t = 0$, $s = 1$ gives the following corollary.

Corollary 2.3. *If $\bar{r}$ is a simple root of $\bar{F}$, then r is uniquely liftable to a root of F.*

Our algorithm will consist of a sequence of invertible changes of variables until we can apply Hensel's lemma or this corollary, at which point we have a root of the transformed system and, by inverting the changes of variables, the original system.

3 Newton polytopes and dual polyhedra

In this section, $F(x) \in K[x_1, \ldots, x_n]$ denotes a *single* multivariate polynomial. Defining $x^e = \prod_i x_i^{e_i}$, it may be written

$$F(x) = \sum_e F_e x^e.$$

Definition 3.1. The **Newton polytope of** F is the lower convex hull of

$$\{(e_1, \ldots, e_n, v(F_e)) \,:\, F_e \neq 0\} \subset \mathbb{Q}^{n+1}.$$

It can be viewed as (the graph of) a convex function

$$\Delta : \Delta^* \to \mathbb{Q}$$

where Δ^* is the convex hull of $\{e \,:\, F_e \neq 0\} \subset \mathbb{Q}^n$.

Each piecewise linear component of Δ is a **face**, and can be viewed as a restriction $P = \Delta|_{P^*}$. We define $E_P = \{e \in P^* \cap \mathbb{Z}^n : v(F_e) = \Delta(e)\}$ to be the exponents of terms of F which lie in P.

Given $s \in \mathbb{Q}^n$, then we define $\Delta_s : \Delta^* \to \mathbb{Q}$ by $\Delta_s(e) = \Delta(e) + s \cdot e$. Defining $P_s^* = \{e \in \Delta^* : \Delta_s(e) \text{ is minimized}\}$ then $P_s = \Delta|_{P_s^*}$ is a face of Δ.

Remark 3.2. In some contexts, such as when working over non-local fields, Δ^* itself is called a Newton polytope (e.g. [55]).

Lemma 3.3. *(a) If $r \in \bar{K}^n$ then the Newton polytope of $F(rx)$ is $\Delta_{v(r)}$. (b) If r is a root of F then* $\dim P_{v(r)} > 0$.

Proof. (a) Indeed $F(rx) = \sum_e (F_e r^e) x^e$ and $v(F_e r^e) = v(F_e) + v(r) \cdot e$. (b) By the ultrametric property applied to the terms of $F(r) = \sum_e F_e r^e = 0$ then $v(F_e) + v(r) \cdot e$ is minimized at least twice, and so $P^*_{v(r)}$ contains at least two points. □

Definition 3.4. If $P = \Delta|_{P^*}$ is a face of Δ, then its **dual polyhedron** is the set of $w \in \mathbb{Q}^n$ such that

$$\Delta_w(e) \leq \Delta_w(e')$$

for all $e \in P^*$, $e' \in \Delta^*$, with equality if and only if $e' \in P^*$. The dual polyhedron of P_s is denoted H_s. The union of dual polyhedra of faces of non-zero dimension is called the **dual variety of** Δ.

Remark 3.5. The dual variety is in fact a tropical variety [39]. We do not exploit this structure in our algorithm.

Lemma 3.6. *(a) The dual polyhedra of different faces are disjoint. (b) A face is orthogonal to its dual polyhedron. (c) If r is a root of F then $v(r) \in H_{v(r)}$.*

Proof. (a) Suppose $P_1 \neq P_2$ and so without loss of generality $P_1 \not\subseteq P_2$ so there exists $e_1 \in P_1 \setminus P_2$ and $e_2 \in P_2$. Suppose w is an element of both dual polyhedra. Then $\Delta_w(e_1) \leq \Delta_w(e_2)$ and $\Delta_w(e_2) < \Delta_w(e_1)$ which is a contradiction.

(b) If P is a face with dual H and $e, e' \in P^*$ then by definition

$$\Delta(e') - \Delta(e) = w \cdot (e - e')$$

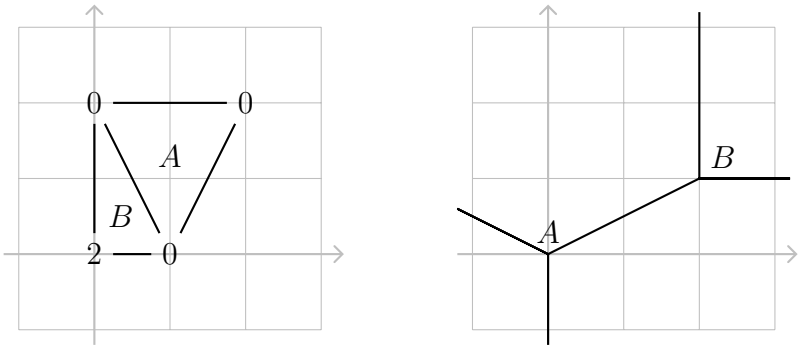

Figure 1: The Newton polytope of $F(x, y) = 4 + x + 2xy + y^2 + xy^2 \in \mathbb{Q}_2[x, y]$ and its dual variety.

for $w \in H$ and so if $w, w' \in H$ then

$$(w - w') \cdot (e - e') = 0.$$

(c) $H_{v(r)}$ is the set of w such that Δ_w is minimized precisely on $P^*_{v(r)}$, which is true for $w = v(r)$. □

Example 3.7. Consider $F(x, y) = 4 + x + 2xy + y^2 + x^2y^2 \in \mathbb{Q}_2[x, y]$. Its Newton polytope is given in Figure 1 alongside its dual variety. For the Newton polytope Δ, we draw the domain P^* of each of its faces P (there are 2 of dimension 2, 5 of dimension 1, 4 of dimension 0) and give the value $\Delta(e)$ at its vertices. The two faces of dimension 2 are labelled A and B, and their dual polyhedra are also labelled. Observe that $v(F_{1,1}) = 1 > \Delta(1, 1) = 0$ so the xy term of F does not contribute to Δ, and so $E_A = \{(1, 0), (0, 2), (1, 2)\}$ and $E_B = \{(0, 0), (1, 0), (0, 2)\}$. ◊

4 Residual systems

Definition 4.1. If P is a face of the Newton polytope of F, then its **leading terms** are the polynomial

$$\operatorname{lead}_P F = \sum_{e \in E_P} F_e^{\#} x^e.$$

The **content** of F is the term

$$\operatorname{cont} F = \pi^{\min_e v(F_e)} x_1^{\min_e e_1} \cdots x_n^{\min_e e_n}$$

and we define $F^{\#} = F/\operatorname{cont} F$.

The **proto-residual polynomial of** P is

$$\overline{(\operatorname{lead}_P F)^{\#}} \in \mathbb{F}[x_1, \ldots, x_n].$$

Lemma 4.2. *Let R_0 be the proto-residual polynomial of P, with dual polyhedron H and suppose r is a root of F such that $v(r) \in H$. Let $\delta = r/\pi^{v(r)}$. Then $R_0(\bar{\delta}) = 0$.*

Proof. Since

$$0 = F(r) = \sum_e (F_e \pi^{v(r)\cdot e})\delta^e,$$

letting $w = \min_e v(F_e) + v(r) \cdot e$ then

$$0 = \overline{\sum_e F_e \pi^{v(r)\cdot e - w}\delta^e} = \sum_{e\,:\,v(F_e)+v(r)\cdot e = w} \overline{F_e^{\#}\delta^e} = \overline{\operatorname{lead}_P F}(\bar{\delta}).$$

Since $\overline{\operatorname{lead}_P F}$ and R_0 differ only by a monomial multiplier, the result follows. □

We now consider a system $F = (F_1, \ldots, F_m) \in K[x_1, \ldots, x_n]^m$ of m polynomials in n variables, and fix faces $P_j : P_j^* \to \mathbb{Q}$ of the Newton polytopes $\Delta_j : \Delta_j^* \to \mathbb{Q}$ with dual polyhedra H_j and intersection $H = \bigcap_i H_i$.

For any polytope P we let $\langle P \rangle = \langle e - e' : e, e' \in P \rangle$ denote the vector space parallel to P.

Definition 4.3. The faces $P_1, \ldots, P_m$ are **axis aligned** if $\langle P_1^*, \ldots, P_m^* \rangle = \mathbb{Q}^d \times \{0\}^{n-d}$ for some $1 \le d \le n$.

In this situation, since each H_j is orthogonal to P_j then $H = \{\hat{w}\} \times \hat{H}$ for some $\hat{w} \in \mathbb{Q}^d$ and $\hat{H} \subset \mathbb{Q}^{n-d}$.

Definition 4.4. If $\hat{w} \in \mathbb{Z}^d$ then the faces are **integrally axis aligned**. If $\hat{w} \in \mathbb{Q} \times \mathbb{Z}^{d-1}$ then the faces are **almost integrally axis aligned**.

Lemma 4.5. *Let $R_0(x) \in \mathbb{F}[x_1, \ldots, x_n]^m$ be the system of proto-residual polynomials of $P_1, \ldots, P_m$. If the faces are axis aligned then $R_0(x) \in \mathbb{F}[x_1, \ldots, x_d]^m$.*

Proof. Fix $e' \in P_j^*$. For any $e \in P_j^*$ we have $e' - e \in \mathbb{Q}^d \times \{0\}^{n-d}$ and therefore $e'_i = e_i$ for all $d < i \leq n$. Now $\operatorname{lead}_{P^* j} F_j$ only includes terms for $e \in E_{P_j} \subset P_j^*$, and so is of the form $G(x_1, \ldots, x_d) x_{d+1}^{e'_{d+1}} \cdots x_n^{e'_n}$. Hence $(\operatorname{lead}_{P_j} F_j)^{\#} = G^{\#} \in K[x_1, \ldots, x_d]$. □

Lemma 4.6. *Suppose that the faces are almost integrally axis aligned, and $\hat{w}_1 = a/b$ in lowest terms. Let $R_0 \in \mathbb{F}[x_1, \ldots, x_d]^m$ be the system of proto-residual polynomials. Then $R_0(x) = R(x_1^b, x_2, \ldots, x_d)$ for some $R \in \mathbb{F}[x_1, \ldots, x_d]^m$.*

Proof. Fix $e' \in E_{P_j}$, then for all $e \in E_{P_j}$ we have

$$v(F_{j,e}) + v(r) \cdot e = v(F_{j,e'}) + v(r) \cdot e'$$

so in particular

$$\tfrac{a}{b}(e_1 - e'_1) = v(F_{j,e'}) - v(F_{j,e}) - \sum_{1<i\leq d} \hat{w}_i (e_i - e'_i) \in \mathbb{Z}$$

and so $e_1 \equiv e'_1 \mod b$. □

Definition 4.7. The **residual system of** $(P_1, \ldots, P_n)$ is $R(x)$.

Lemma 4.8. *If r is a root with $v(r) \in H$, then*

$$R\left(\overline{r_1^b/\pi^a}, \overline{r_2/\pi^{\hat{w}_2}}, \ldots, \overline{r_d/\pi^{\hat{w}_d}}\right) = 0.$$

In particular, the residue class field of $K(r_1, \ldots, r_d)$ contains a root of R.

Proof. Follows trivially from Lemma 4.2 and Definition 4.7. □

5 Change of variables

In this section, we show that given faces $P_1, \ldots, P_m$ there is always an invertible change of variables such that the transformed faces are almost integrally axis aligned, and therefore we can always define residual systems.

Lemma 5.1. *For $M \in \mathrm{GL}_n(\mathbb{Z})$, define $(x^M)_j = \prod_i x_i^{M_{i,j}}$. (a) Then $x \mapsto x^M$ is an action of $\mathrm{GL}_n(\mathbb{Z})$ on the set of all monomials $x^{\mathbb{Z}^n}$. (b) $(x^M)^e = x^{Me}$. (c) For $F(x) \in K[x_1, \ldots, x_n]$, defining $F^M(x) = F(x^M)$ then $F^M_{Me} = F_e$. (d) Its Newton polytope is $\Delta^M : M\Delta^* \to \mathbb{Q} : Me \mapsto \Delta(e)$. (e) If r is a root of F then $r^{M^{-1}}$ is a root of F^M. (f) If H is the dual polyhedron of face $P : P^* \to \mathbb{Q}$, then MP^* is a face of Δ^M and $M^{-1}H$ is its dual polyhedron.*

Proof. (a) $(x^M)^N_k = \prod_j (x^M)_j^{N_{j,k}} = \prod_j (\prod_i x_i^{M_{i,j}})^{N_{j,k}} = \prod_i x_i^{\sum_j M_{i,j}N_{j,k}} = \prod_i x_i^{(MN)_{i,k}} = (x^{MN})_k$. (b) $(x^M)^e = \prod_j (x^M)_j e_j = \prod_j (\prod_i x_i^{M_{i,j}}) e_j = \prod_i x_i^{\sum_j M_{i,j}e_j} = x^{Me}$. (c) $F^M(x) = F(x^M) = \sum_e F_e x^{Me}$. (d) Follows from (c). (e) $F^M(r^{M^{-1}}) = F((r^{M^{-1}})^M) = F(r) = 0$. (f) Follows from (d) and (e). □

We now fix a system $F(x) \in K[x_1, \ldots, x_n]^m$ and faces $P_1, \ldots, P_m$ of the Newton polytopes $\Delta_1, \ldots, \Delta_m$ with duals $H_1, \ldots, H_m$ with intersection $H = \bigcap_i H_i$. Let $d = \dim \langle P_1, \ldots, P_m \rangle$.

Lemma 5.2. *There exists $M \in \mathrm{GL}_n(\mathbb{Z})$ such that $MP_1^*, \ldots, MP_n^*$ are axis aligned.*

Proof. Choose a basis $\{a_1, \ldots, a_d\} \subset \langle P_1, \ldots, P_m \rangle$. By multiplying by a suitable integer, we may assume $a_i \in \mathbb{Z}^n$. Let $A = (a_1 \ \ldots \ a_d) \in \mathbb{Z}^{n \times d}$.

Let $B_0 = (b_1 \ \ldots \ b_d) \in \mathbb{Z}^{n \times d}$ be a $\mathbb{Q}$-saturation of A (i.e. satisfying $\langle a_1, \ldots, a_d \rangle_{\mathbb{Q}} = \langle b_1, \ldots, b_d \rangle_{\mathbb{Q}}$ and $\langle b_1, \ldots, b_d \rangle_{\mathbb{Z}} = \langle a_1, \ldots, a_d \rangle_{\mathbb{Q}} \cap \mathbb{Z}^n$; equivalently $\det(B_0^T B_0) = \pm 1$).

Then $\mathbb{Z}^n / \langle b_1, \ldots, b_d \rangle_{\mathbb{Z}} \cong \mathbb{Z}^{n-d}$ (as $\mathbb{Z}$-modules). Let $b_{d+1}, \ldots, b_n \in \mathbb{Z}^n$ be the preimages of the standard basis for $\mathbb{Z}^{n-d}$ under such an isomorphism and let $B = (b_1 \ \ldots \ b_n) \in \mathbb{Z}^{n \times n}$.

Then by construction $\mathbb{Z}^n / \langle b_1, \ldots, b_n \rangle_{\mathbb{Z}} \cong \mathbb{Z}^0$ so $B \in \mathrm{GL}_n(\mathbb{Z})$. Let $M = B^{-1}$. Fix $e'_j \in P_j^*$, then by construction if $e \in P_j^*$ then $e - e'_j \in \langle b_1, \ldots, b_d \rangle_{\mathbb{Q}}$, so $M(e - e'_j) \in \mathbb{Q}^d \times \{0\}^{n-d}$, and so $MP_j^* \subset Me'_j + \mathbb{Q}^d \times \{0\}^{n-d}$. □

Lemma 5.3. *There exists $M \in \mathrm{GL}_n(\mathbb{Z})$ such that $MP_1^*, \ldots, MP_n^*$ are almost integrally axis aligned.*

Proof. By Lemma 5.2, without loss of generality they are axis aligned. Then the intersection of the dual polyhedra is $H = \{\hat{w}\} \times \hat{H}$ for $\hat{w} \in \mathbb{Q}^d$ and $H \subset \mathbb{Q}^{n-d}$.

Observe that if $M_0 \in \mathrm{GL}_d(\mathbb{Z})$ and we extend it with 1 on the diagonal to $M \in \mathrm{GL}_n(\mathbb{Z})$ then $MP_1^*, \ldots, MP_n^*$ are still axis aligned, and $M^{-1}H = \{M_0^{-1}\hat{w}\} \times \hat{H}$. It remains to find M_0 such that $M_0^{-1}\hat{w} \in \mathbb{Q} \times \mathbb{Z}^{d-1}$.

Write $\hat{w}_i = a_i/c_i$ in lowest terms, let $c = \mathrm{lcm}_i\, c_i$ and find $b_1 \in \mathbb{Z}^d$ such that $b_1 \cdot \hat{w} = 1/c$.

Let $g = \gcd_i b_{1,i}$, then $1/gc = \sum_i (b_{1,i}/g) w_i \in \mathbb{Z}/c$, and so $g = 1$. Hence the matrix with the single column b_1 is saturated and so $\mathbb{Z}^d/\langle b_1 \rangle \cong \mathbb{Z}^{d-1}$. Let $b_2^*, \ldots, b_d^*$ be preimages under such an isomorphism of the standard basis for $\mathbb{Z}^{d-1}$.

Write $b_i^* \cdot \hat{w} = h_i/c$ and let $b_i = b_i^* - h_i b_1$. Then $b_i \cdot \hat{w} = 0$ and $b_i \equiv b_i^* \mod b_1$, so $b_2, \ldots, b_d$ are still the standard basis for $\mathbb{Z}^{d-1}$, and hence $B_0 = (b_1 \ldots b_d) \in \mathrm{GL}_d(\mathbb{Z})$.

Let $M_0 = B_0^{-1}$. By construction $M_0^{-1}\hat{w} = B_0\hat{w} = (1/c, 0, \ldots, 0) \in \mathbb{Q} \times \mathbb{Z}^{d-1}$ as required. □

6 Algorithm I

We now present our root-finding algorithm. The inputs b and B exist mainly for recursion; to find all roots set $b = 0$ and $B = \mathbb{Q}^n$. One can think of b as an approximation to a root of F, and B as measuring how good that approximation is. The proof of Lemma 6.2 motivates the steps of the algorithm.

Algorithm 6.1. Given a square system $F(x) \in K[x_1, \ldots, x_n]^n$ without repeated roots, a vector $b \in K^n$, and an open polyhedron $B \subseteq \mathbb{Q}^n$, returns the roots r of F in K^n such that $v(r - b) \in B$.

1: Let $m = v(b) \in \mathbb{Z}^n$.
2: Let $G(x) = F(\pi^m x)^\# \in \mathcal{O}[x_1, \ldots, x_n]^n$.
3: Let $t = v(J(G)(\pi^{-m}b))$.
4: Let $A = \mathbb{Q}_{\geq t+m_1+1} \times \cdots \times \mathbb{Q}_{\geq t+m_n+1}$.
5: **if** $\min v(G(\pi^{-m}b)) > 2t$ and $B \subset A$ **then**
6: Hensel lift $\pi^{-m}b$ to a root $\pi^{-m}r$ of G.
7: **if** $v(r - b) \in B$ **then**
8: Return $\{r\}$.
9: **else**

10: Return $\emptyset$.
11: **end if**
12: **end if**
13: **for** $i = 1, \dots, n$ **do**
14: Let $\Delta_i : \Delta_i^* \to \mathbb{Q}$ be the Newton polytope of $F_i(x + b)$.
15: Let $\mathcal{P}_i$ be the set of its faces.
16: Let $\mathcal{H}_i = \{H \cap B \,:\, P \in \mathcal{P}_i, H \text{ dual to } P\} \setminus \{\emptyset\}$.
17: **end for**
18: Let $\mathcal{H} = \{\bigcap_{i=1}^n H_i \,:\, H_i \in \mathcal{H}_i\} \setminus \{\emptyset\}$.
19: Let $S = \emptyset$.
20: **for** $H \in \mathcal{H}$ **do**
21: Let $H_i \in \mathcal{H}_i$ uniquely such that $\bigcap_i H_i = H$.
22: Let $P_i \in \mathcal{P}_i$ be the corresponding faces.
23: Use the proof of Lemma 5.2 to find $M \in \mathrm{GL}_n(\mathbb{Z})$ such that $M^{-1}H = \{\hat{w}\} \times \hat{H}$ for some $\hat{w} \in \mathbb{Q}^d$ and $\hat{H} \subset \mathbb{Q}^{n-d}$. In particular MP_i^* is the domain of a face of the Newton polygon of $F_i(x^M + b)$.
24: **if** $\hat{w} \in \mathbb{Z}^d$ **then**
25: Let $R(x) \in \mathbb{F}[x_1, \dots, x_d]^n$ be the residual system.
26: **for** roots $\bar{\delta} \in \mathbb{F}^d$ of R **do**
27: Let $b' = b + (\pi^{\hat{w}_1}\delta_1, \dots, \pi^{\hat{w}_d}\delta_d, 0, \dots, 0)^M$.
28: Let $B' = M(\mathbb{Q}_{>\hat{w}_1} \times \cdots \times \mathbb{Q}_{>\hat{w}_d} \times \hat{H})$.
29: Recursively let S' be the roots r of F such that $v(r - b') \in B'$.
30: Let $S = S \cup S'$.
31: **end for**
32: **end if**
33: **end for**
34: Return S.

Lemma 6.2. *If this algorithm terminates, then it returns the set of all roots $r \in K^n$ of F such that $v(r - b) \in B$.*

Proof. Lines 1 to 12: This is the terminating condition. By Hensel's lemma with $s = 2t + 1$, if $\min v(G(\pi^{-m}b)) > 2t$ then there is a unique root $\pi^{-m}r$ of G such that $\min v(\pi^{-m}b - \pi^{-m}r) \geq t + 1$. Equivalently, there is a unique root r of F such

that $v(r-b) \in A$. If also $B \subset A$ then there is at most one root of F such that $v(r-b) \in B$, and so we return either $\{r\}$ or $\emptyset$.

If the terminating condition does not hold, we proceed to improve our bounds b and B on the roots of F.

Lines 13 to 17: Observe that r is a root of $F_i(x)$ if and only if $r-b$ is a root of $F_i(x+b)$, and hence the dual variety of Δ_i gives the possible values of $v(r-b)$. We conclude that if r is a root of F_i and $v(r-b) \in B$ then there exists a unique $H \in \mathcal{H}_i$ such that $v(r-b) \in H$.

Line 18: It follows that if r is a root of F and $v(r-b) \in B$ then there exists a unique $H \in \mathcal{H}$ such that $v(r-b) \in H$.

Line 20: In this for loop, we restrict attention to just those roots such that $v(r-b) \in H$. By the preceding comments, we will consider all roots precisely once at some point in this loop.

Line 23: Having selected $M \in \mathrm{GL}_n(\mathbb{Z})$, then we know that r is a root of F with $v(r-b) \in H$ if and only if $(r-b)^{M^{-1}}$ is a root of $F(x^M+b)$ with

$$v((r-b)^{M^{-1}}) \in M^{-1}H = \{\hat{w}\} \times \hat{H}. \qquad (*)$$

Line 24: In particular $r, b \in K^n$ and so $v((r-b)^{M^{-1}}) \in \mathbb{Z}^n$ and so $\hat{w} \in \mathbb{Z}^d$. Hence we safely ignore cases where $\hat{w} \notin \mathbb{Z}^d$.

Line 26: In this for loop, we restrict attention to just the roots such that

$$\overline{(r-b)_{1\ldots d}^{M^{-1}}/\pi^{\hat{w}}} = \bar{\delta}. \qquad (**)$$

By Lemma 4.8

$$R\left(\overline{(r-b)_{1\ldots d}^{M^{-1}}/\pi^{\hat{w}}}\right) = 0$$

and therefore we will consider all roots precisely once at some point in this loop.

Lines 27 to 30: The conditions $(*)$ and $(**)$ are precisely equivalent to

$$v((r-b)^{M^{-1}} - (\pi^{\hat{w}_1}\delta_1, \ldots, \pi^{\hat{w}_d}\delta_d, 0, \ldots, 0)) \in \mathbb{Q}_{>\hat{w}_1} \times \cdots \times \mathbb{Q}_{>\hat{w}_d} \times \hat{H}$$

which in turn is precisely equivalent to $v(r-b') \in B'$. We deduce that in Line 29 we find all roots currently under consideration, which we accumulate into S. $\square$

Remark 6.3. In the terminating condition, it suffices to find any $m \in \mathbb{Z}^n$ such that $\pi^{-m}b \in \mathcal{O}^n$, $\min v(G(\pi^{-m}b)) > 2t$ and $B \subset A$. Our choice of m appears to be sufficient to guarantee the algorithm terminates (cf. Conjecture 6.6).

Remark 6.4. Note that the residual system $R(x)$ does not necessarily define a zero-dimensional variety, and therefore can have many roots in $\mathbb{F}^d$. Therefore, this algorithm is only suitable when $|\mathbb{F}|^n$ is small, and is only possible because the residue class field of K happens to be finite.

Worse still, it is unlikely that this algorithm can be generalized to factoring, because factoring can be viewed as root-finding in $\bar{K}$, whose residue class field $\bar{\mathbb{F}}$ is not finite. We deal with this in §7.

Remark 6.5. Instead of requiring $\hat{w} \in \mathbb{Z}^d$, we can perform the stricter check $H \cap \mathbb{Z}^n \neq \emptyset$. Furthermore, we may replace H by the convex hull of $H \cap \mathbb{Z}^n$, which may have a lower dimension.

Conjecture 6.6. *Assuming F does not have repeated roots, this algorithm terminates.*

Justification. Suppose the algorithm does not terminate for some inputs F, b and B. Since each loop is finite, this can only occur if there is infinitely nesting recursion. Fix a single infinite sequence of nested calls to the algorithm, and let $(b, B) = (b_0, B_0), (b_1, B_1), \ldots$ be the parameters in each call.

Observe that if $H \subset B_i$ in one iteration, with $M^{-1}H = \{\hat{w}\} \times \hat{H}$, then $M^{-1}B_{i+1} = \mathbb{Q}_{>\hat{w}_1} \times \cdots \times \mathbb{Q}_{>\hat{w}_d} \times \hat{H}$ lies strictly above H along each of the first d coordinates. By Lemma 3.3 we have $d > 0$.

This probably implies that $(b_0, b_1, \ldots)$ is a Cauchy sequence, and so has a limit r, and this is necessarily a root of F. In particular, $m = v(b_i)$ is eventually always equal to $v(r)$, and so $G(x) = F(\pi^{-m}x)$ is eventually constant, and so $t = v(J(G)(b_i))$ is eventually constant, and so A is eventually constant. Eventually we will have $\min v(G(\pi^{-m}b)) > 2t$ and probably $B_i \subset A$. Since this is the terminating condition, we deduce that this recursion eventually terminates, which contradicts our initial assumption. □

7 Algorithm II

We present a modified version of the previous algorithm which does not always compute all roots of the residual system, but instead finds the irreducible components of the variety it defines. We therefore learn some non-linear information about some p-adic coefficients of the roots, which is represented as new equations in new variables. Hence, the number of variables increases at each iteration.

Definition 7.1. If Z is a polynomial ideal, then we write $Z(x) = 0$ as shorthand for: $z(x) = 0$ for all $z \in Z$.

Algorithm 7.2. Given a square system $F(x) \in K[x_1, \ldots, x_n]^n$ without repeated roots, an open polyhedron $B \subseteq \{0\}^m \times \mathbb{Q}^{n-m}$ for some $0 \leq m \leq n$, and a finite set $\mathcal{Z}$ of ideals of $\mathbb{F}[x_1, \ldots, x_m]$, returns the roots r of F in K^n such that $v(r) \in B$ and $Z(\bar{r}_{1\ldots m}) \neq 0$ for all $Z \in \mathcal{Z}$.

1: **for** $i = 1, \ldots, n$ **do**
2: Let $\Delta_i : \Delta_i^* \to \mathbb{Q}$ be the Newton polytope of $F_i(x)$.
3: Let $\mathcal{P}_i$ be the set of its faces.
4: Let $\mathcal{H}_i = \{H \cap B \,:\, P \in \mathcal{P}_i, H \text{ dual to } P\} \setminus \{\emptyset\}$.
5: **end for**
6: Let $\mathcal{H} = \{\bigcap_{i=1}^n H_i \,:\, H_i \in \mathcal{H}_i\} \setminus \{\emptyset\}$.
7: Let $S = \emptyset$.
8: **for** $H \in \mathcal{H}$ **do**
9: Let $H_i \in \mathcal{H}_i$ uniquely such that $\bigcap_i H_i = H$.
10: Let $P_i \in \mathcal{P}_i$ be the corresponding faces.
11: Use the proof of Lemma 5.2 to find $M \in \mathrm{GL}_n(\mathbb{Z})$, block diagonal with first block I_m, such that $M^{-1}H = \{\hat{w}\} \times \hat{H}$ for some $\hat{w} \in \{0\}^m \times \mathbb{Q}^{d-m}$ and $\hat{H} \subset \mathbb{Q}^{n-d}$. In particular MP_i^* is the domain of a face of the Newton polygon of $F_i(x^M)$.
12: **if** $\hat{w} \in \mathbb{Z}^d$ **then**
13: Let $R(x) \in \mathbb{F}[x_1, \ldots, x_d]^n$ be the residual system.
14: Let $J_1, \ldots, J_t$ be the prime decomposition of $\langle R \rangle$.
15: **for** $i = 1, \ldots, t$ **do**

16: Let $J = \langle \bar{G} \rangle = J_i$ for some $G \in \mathcal{O}[x_1, \ldots, x_d]^k$.
17: Let $\mathcal{J} = \{J_j : j < i\}$.
18: **if** $\dim J = 0$ **then**
19: **if** $\bar{G}$ has one root **then**
20: Let $\bar{\delta} \in \mathbb{F}^d$ be the root.
21: **if** $Z(\bar{\delta}_{1\ldots m}) = 0$ for some $Z \in \mathcal{Z}$ **then**
22: Go to next i.
23: **else if** $d = n$ and $\mathrm{Jac}(R)(\bar{\delta}) \neq 0$ **then**
24: Hensel lift δ to a root of $F((\pi^{\hat{w}}x)^M)^{\#}$, and hence a root r of F.
25: $S = S \cup \{r\}$.
26: Go to next i.
27: **end if**
28: **else**
29: Go to next i.
30: **end if**
31: **end if**
32: Let $m \in \mathcal{O}[y_1, \ldots, y_k]^d$ such that $R = \bar{m}(\bar{G})$.
33: Let $C_j = \mathrm{cont}\,\mathrm{lead}_{MP_j} F_j(x^M)$ for $j = 1, \ldots, n$.
34: Let $F'(x,y) = \begin{pmatrix} F((\pi^{\hat{w}}x)^M) - C(x) \cdot (m(G(x_{1\ldots d})) - m(y)) \\ y - G(x_{1\ldots d}) \end{pmatrix} \in K[x,y]^{n+k}$.
35: Let $B' = \{0\}^d \times \hat{H} \times \mathbb{Q}^k_{>0}$.
36: Let $\mathcal{Z}' = \mathcal{J} \cup \{Z\mathbb{F}[x_1, \ldots, x_d] : Z \in \mathcal{Z}\}$.
37: Recursively let S' be the set of roots r of F' such that $v(r) \in B'$ and $Z(\bar{r}_{1\ldots d}) \neq 0$ for all Z in $\mathcal{Z}'$.
38: Let $S = S \cup \{(\pi^{\hat{w}} r_{1\ldots n})^M : r \in S'\}$.
39: **end for**
40: **end if**
41: **end for**
42: Return S.

Lemma 7.3. *If this algorithm terminates, then it returns the set of all roots $r \in K^n$ of F such that $v(r-b) \in B$.*

Proof. Lines 1 to 13: This is essentially identical to the previous algorithm. We compute Newton polytopes, branch over intersections of dual polyhedra (restricting to just the roots with corresponding valuations), perform a change of variables, and check the valuation is integral.

Lines 14 to 17: We know that $R(\bar{r}_{1\ldots d}/\pi^{\hat{w}}) = 0$ and therefore there is some unique $1 \leq i \leq d$ such that if $1 \leq j \leq i$ then $J_j(\bar{r}_{1\ldots d}/\pi^{\hat{w}}) = 0$ if and only if $i = j$. Hence, we loop over the possibilities for i and restrict attention to the corresponding roots, namely those such that $J(\bar{r}_{1\ldots d}/\pi^{\hat{w}}) = 0$ and $Z(\bar{r}_{1\ldots d}/\pi^{\hat{w}}) \neq 0$ for $Z \in \mathcal{J}$.

Lines 18 to 31: This is the terminating condition. If $\dim J = 0$ then J has finitely many roots in $\bar{\mathbb{F}}^d$; since J is prime, these roots are distinct and Galois conjugate. Hence, if J has multiple roots, then these do not lie in $\mathbb{F}^d$, and therefore we can safely discard this case. If $\bar{\delta}$ is the root and $Z(\bar{\delta}_{1\ldots m}) = 0$ for some $Z \in \mathcal{Z}$, then noting that $\bar{\delta}_{1\ldots m} = \bar{r}_{1\ldots m}$ we may safely discard this case. Otherwise if $d = n$ and $J(R)(\bar{\delta}) \neq 0$ then $\bar{\delta}$ is a simple root of $R(x) = \overline{F((\pi^{\hat{w}}x)^M)^{\#}}$ and so by Corollary 2.3, δ lifts uniquely to a root of $F((\pi^{\hat{w}}x)^M)^{\#}$ and hence we find a root r of F, which we include in S and then terminate this case.

Lines 32 to 38: The map $r \mapsto (r^{M^{-1}}/\pi^{\hat{w}}, G(r^{M^{-1}}/\pi^{\hat{w}}))$ gives a 1-1 correspondence between roots of F and roots of F'. The roots S' of F' are precisely those corresponding to the roots of F currently under consideration. Hence, we include all roots of F currently under consideration in S. □

Conjecture 7.4. *Assuming F does not have repeated roots, this algorithm terminates.*

Justification. The key feature is our construction of $F'(x, y)$. By definition $\bar{m}_j(\bar{G}) = R_j = \overline{(\mathrm{lead}_{MP_j} F_j(x^M))/C_j}$, and $\mathrm{lead}_{MP_j} F_j(x^M)$ is some of the terms of $F_j((\pi^{\hat{w}}x)^M)$, and therefore by subtracting $C_j(x) \cdot m(G(x))$ we cancel out at least the leading p-adic coefficient of each of these leading terms. Hence $y - G(x)$ captures what we already know, and $F((\pi^{\hat{w}}x)^M) - C(x) \cdot (m(G(x)) - m(y))$ allows us to look one p-adic coefficient deeper into the equations. □

There are numerous ways we can try to make this more efficient.

Remark 7.5. If $\dim J = 0$ and $\bar{G}$ has one root $\bar{\delta}$ which is not Hensel liftable, then

instead of producing a new system with $k = d$ extra variables, we can perform a linear change of variables $x_i \mapsto x_i + \bar{\delta}$ just as in Algorithm I.

Remark 7.6. If J contains a polynomial of the form $U(x^e)$ where U is univariate and irreducible, then either $\deg U > 1$ and so J has no roots in $\mathbb{F}^d$ and we can terminate this branch, or else we can perform a linear change of variables $x^e \mapsto x^e + \delta$ where $\bar{\delta}$ is the root of U as in the previous remark.

Remark 7.7. We can reduce $\mathcal{Z}$ as follows: if there exists $Z \in \mathcal{Z}$ such that $Z \triangleleft \langle \mathcal{Z} \setminus Z \rangle$, then we can remove Z from $\mathcal{Z}$.

Similarly, if we explicitly track an ideal $S \triangleleft \mathbb{F}[x_1, \ldots, x_m]$ such that $S(\bar{r}_{1\ldots m}) = 0$, and find $Z \in \mathcal{Z}$ such that $\langle Z, S \rangle = \langle 1 \rangle$ then Z can be removed from $\mathcal{Z}$. When recursing, we include J in S.

Remark 7.8. There is a choice of generating set $\bar{G}$ for J, which will affect k. Typically $\bar{G}$ will be a Gröbner basis.

Remark 7.9. As this algorithm iterates, we add more new polynomials to our system. These polynomials are known exactly and have a particular form, so we should be able to compute Newton polytopes etc. more efficiently than for generic polynomials.

8 Worked example

Consider the system

$$F(x) = \begin{pmatrix} x^2 + x + y + 27 \\ x^2 + 4x + y - 3 \end{pmatrix} \in \mathbb{Q}_3[x, y]^2$$

which by considering $F_2 - F_1 = 3(x - 10)$ has the single root $(10, -137) \in \mathbb{Q}_3^2$ of valuation $(0, 0)$. We now use Algorithm 7.2 to draw the same conclusion.

Definition 8.1. We use the notation $\langle p_1, \ldots, p_k \rangle$ to denote the interior of the convex hull of $\{p_1, \ldots, p_k\}$, for $p_i \in \mathbb{Q}^n$. We use $\langle p_1, \ldots, p_k : d_1, \ldots, d_\ell \rangle$ to denote the open polyhedron $\langle p_1, \ldots, p_k \rangle + d_1\mathbb{Q}_{>0} + \ldots + d_\ell\mathbb{Q}_{>0}$.

Computing Newton polytopes and dual polyhedra (Figure 2), we deduce

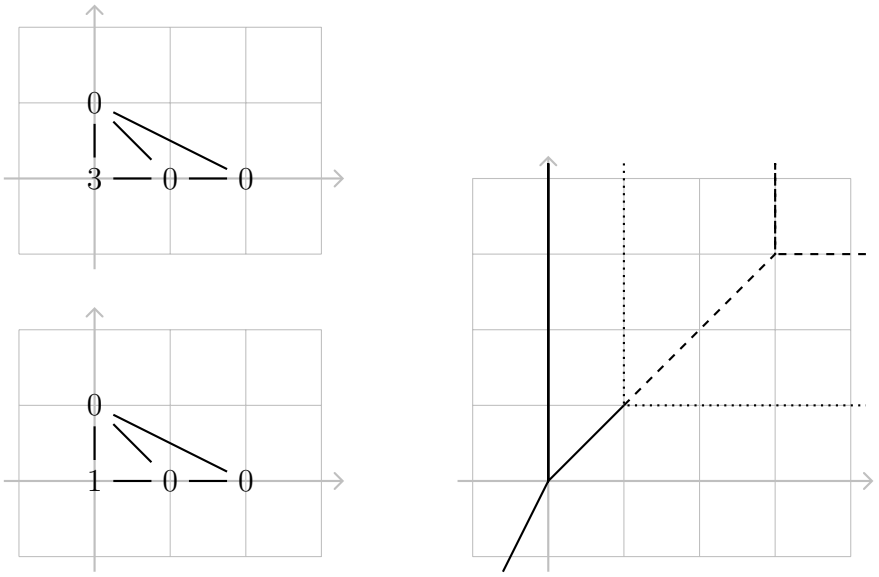

Figure 2: Newton polytopes and dual varieties of $F(x, y)$. Dual variety of F_1 is dashed, F_2 is dotted, and the intersection is solid.

---

that $v(r)$ is in one of: $H_1 = \langle(0,0)\rangle$, $H_2 = \langle(1,1)\rangle$, $H_3 = \langle(0,0),(1,1)\rangle$, $H_4 = \langle(0,0):(0,1)\rangle$, $H_5 = \langle(0,0):(-1,-2)\rangle$.

Case 1: $v(r) \in H_1$**.** Then $v(r) = (0,0)$ and we get the residual system

$$R(x) = \overline{F(x,y)} = \begin{pmatrix} x^2 + x + y \\ x^2 + x + y \end{pmatrix} \in \mathbb{F}_3[x,y]^2$$

and $\langle R\rangle$ is prime already.

We take $G(x,y) = x^2 + x + y \in \mathbb{Q}_3[x,y]$, so that $\langle \bar{G}\rangle = \langle R\rangle$, and take $m(z) = (z,z) \in \mathbb{Q}_3[z]^2$ so that $\bar{m}(\bar{G}) = R$.

We construct the new system

$$F'(x,y,z) = (F(x) - m(G) + m(z), z - G) = \begin{pmatrix} z + 27 \\ z + 3x - 3 \\ z - x^2 - x - y \end{pmatrix} \in \mathbb{Q}_3[x,y,z]^3$$

and recursively look for its roots r' such that $v(r') \in B' = \{0\}^2 \times \mathbb{Q}_{>0}$.

From the dual variety, we deduce that the only possibility is $v(r') = (0, 0, 3)$, giving the residual system

$$R'(x, y, z) = \overline{F'(x, y, 27z)^{\#}} = \begin{pmatrix} z + 1 \\ x + 2 \\ x^2 + x + y \end{pmatrix} \in \mathbb{F}_3[x, y, z]^3.$$

Now $\langle R' \rangle = \langle x + 2, y + 2, z + 1 \rangle$ is already prime, zero dimensional, and has a single root $(1, 1, 2) \in \mathbb{F}_3^3$. We can apply Corollary 2.3 to deduce there is a unique root r' of F' close to $(1, 1, 27 \cdot 2) \in \mathbb{Q}_3^3$, the first two coordinates of which give a unique root r of $F(x, y)$ close to $(1, 1)$.

Case 2: $v(r) \in H_2$**.** Hence $v(r) = (1, 1)$ and we get the residual system

$$R(x) = \overline{F(3x, 3y)^{\#}} = \begin{pmatrix} x + y \\ x + y + 2 \end{pmatrix} \in \mathbb{F}_3[x, y]^2.$$

But $\langle R \rangle = \langle 1 \rangle$ and so this has no roots.

Case 3: $v(r) \in H_3$**.** As in Remark 6.5, we observe that $H_3 \cap \mathbb{Z}^2 = \emptyset$ to deduce there are no roots in this case.

Case 4: $v(r) \in H_4$**.** That is, $v(r_1) = 0$ and $v(r_2) > 0$. The residual system is $(x + 1, x + 1)$ and so we produce the new system

$$F'(x, y, z) = \begin{pmatrix} F_1 - x(x + 1 - z) \\ F_2 - x(x + 1 - z) \\ z - x - 1 \end{pmatrix} = \begin{pmatrix} y + 27 + xz \\ 3x + y - 3 + xz \\ z - x - 1 \end{pmatrix}$$

and look for its roots r' with valuation in $B' = \{0\} \times \mathbb{Q}_{>0}^2$.

Considering dual varieties, we deduce $v(r')$ is in one of $H_1' = \langle (0, 1, 1) \rangle$, $H_2' = \langle (0, 3, 3) \rangle$, $H_3' = \langle (0, 0, 0), (0, 1, 1) \rangle$, $H_4' = \langle (0, 1, 1), (0, 3, 3) \rangle$, $H_5' = \langle (0, 3, 3) : (0, 1, 0) \rangle$, $H_6' = \langle (0, 3, 3) : (0, 0, 1) \rangle$.

Case 4.1: $v(r') \in H'_1$**.** That is, $v(r') = (0, 1, 1)$, giving residual system $(xz + y, xz + x + y + 2, x + 1)$ which has no roots.

Case 4.2: $v(r') \in H'_2$**.** That is, $v(r') = (0, 3, 3)$, giving residual system $(xz + y + 1, x + 2, x + 1)$ which has no roots.

Case 4.3: $v(r') \in H'_3$**.** This contains no integral points, so there are no such roots.

Case 4.4: $v(r') \in H'_4$**.** Note that $H'_4 \cap \mathbb{Z}^2 = \{(0, 2, 2)\}$ so $v(r') = (0, 2, 2)$, giving residual system $(xz + y, x + 2, x + 1)$ which has no roots.

Case 4.5: $v(r') \in H'_5$**.** This gives the residual system $(xz+1, x+2, x+1)$ which has no roots.

Case 4.6: $v(r') \in H'_6$**.** This gives the residual system $(y + 1, x + 2, x + 1)$ which has no roots.

Case 5: $v(r) \in H_5$**.** With $M = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 2 \end{smallmatrix}\right)$, we get the residual system $(x + 1, x + 1)$ and produce

$$F'(x, y, z) = \begin{pmatrix} F_1(y, xy^2) - y^2(x + 1 - z) \\ F_2(y, xy^2) - y^2(x + 1 - z) \\ z - x - 1 \end{pmatrix} = \begin{pmatrix} y + 27 + y^2 z \\ 4y - 3 + y^2 z \\ z - x - 1 \end{pmatrix}$$

and look for roots r' of F' with $v(r') \in \{0\} \times \mathbb{Q}_{<0} \times \mathbb{Q}_{>0}$.

Computing dual varieties, we find that $v(r') \in \langle (0, 0, 0) : (0, -1, 1) \rangle$. With $M = \left(\begin{smallmatrix} 1 & -1 \\ 0 & 1 \end{smallmatrix}\right)$, we get the residual system $(y + 1, y + 1, x + 1)$. We would ordinarily now add two new variables and equations, but as we already have a variable

$z = x + 1$ we only need to add $w = y + 1$ to produce

$$F''(x,y,z,w) = \begin{pmatrix} zF_1'(x,y/z,z) - y(y+1-w) \\ zF_2'(x,y/z,z) - y(y+1-w) \\ F_3'(x,y/z,z) \\ w - y - 1 \end{pmatrix} = \begin{pmatrix} 27z + yw \\ 3y - 3z + yw \\ z - x - 1 \\ w - y - 1 \end{pmatrix}$$

and look for roots r'' of F'' with $v(r'') \in H'' = \{0\}^2 \times \mathbb{Q}^2_{>0}$.

The intersection of the dual varieties is $\langle (0,0,0,3) \rangle \cup \langle (0,0,0,3) : (1,0,0,0) \rangle$ which does not meet H''. We conclude there are no such roots.

We have considered all cases. Only one case led to a root, and we conclude that F has one root r, with $r_1 \equiv r_2 \equiv 1 \mod 3$ and $r_1^2 + r_1 + r_2 \equiv 2 \cdot 3^3 \mod 3^4$, which agrees with $r = (10, -137)$.

Bibliography

[1] E. Artin and J. Tate. *Class field theory*. Reprinted with corrections from the 1967 original. AMS Chelsea Publishing, Providence, RI, 2009, pp. viii+194.

[2] C. Awtrey. "Dodecic Local Fields". PhD thesis. Arizona State University, 2015.

[3] C. Awtrey et al. "Degree 14 2-adic fields". In: *Involve* 8.2 (2015), pp. 329–336. DOI: 10.2140/involve.2015.8.329.

[4] C. Awtrey et al. "Groups of order 16 as Galois groups over the 2-adic numbers". In: *Int. J. Pure and Applied Math.* 103.4 (2015), pp. 781–795. DOI: 10.12732/ijpam.v103i4.13.

[5] C. Awtrey et al. "On Galois groups of degree 15 polynomials". In: *Int. J. Pure and Applied Math.* 104.3 (2015), pp. 407–420. DOI: 10.12732/ijpam.v104i3.10.

[6] D. N. Bernstein. "The number of roots of a system of equations". In: *Funkcional. Anal. i Priložen.* 9.3 (1975), pp. 1–4. DOI: 10.1007/BF01075595.

[7] A. R. Booker et al. "A database of genus-2 curves over the rational numbers". In: *LMS J. Comput. Math.* 19.suppl. A (2016), pp. 235–254. DOI: 10.1112/S146115701600019X.

[8] W. Bosma, J. Cannon, and C. Playoust. "The Magma algebra system. I. The user language". In: *J. Symbolic Comput.* 24.3-4 (1997). Computational algebra and number theory (London, 1993), pp. 235–265. DOI: 10.1006/jsco.1996.0125.

[9] N. Bruin, E. V. Flynn, and D. Testa. "Descent via (3, 3)-isogeny on Jacobians of genus 2 curves". In: *Acta Arith.* 165.3 (2014), pp. 201–223. DOI: 10.4064/aa165-3-1.

[10] X. Caruso. *Computations with p-adic numbers*. 2017. arXiv: 1701.06794.

[11] J. Coates et al. "Root numbers, Selmer groups, and non-commutative Iwasawa theory". In: *J. Algebraic Geom.* 19.1 (2010), pp. 19–97. DOI: 10.1090/S1056-3911-09-00504-9.

[12] K. Conrad. "Recognizing Galois groups S_n and A_n". Lecture notes. URL: https://www.math.uconn.edu/~kconrad/blurbs/galoistheory/galoisSnAn.pdf.

[13] F.-E. Diederichsen. "Über die Ausreduktion ganzzahliger Gruppendarstellungen bei arithmetischer Äquivalenz". In: *Abh. Math. Sem. Hansischen Univ.* 13 (1940), pp. 357–412.

[14] T. Dokchitser. "Computing special values of motivic L-functions". In: *Experiment. Math.* 13.2 (2004), pp. 137–149. DOI: 10.1080/10586458.2004.10504528.

[15] T. Dokchitser. *Models of curves over DVRs*. 2018. arXiv: 1807.00025.

[16] T. Dokchitser and V. Dokchitser. "Quotients of hyperelliptic curves and étale cohomology". In: *Q. J. Math.* 69.2 (2018), pp. 747–768. DOI: 10.1093/qmath/hax053.

[17] T. Dokchitser, V. Dokchitser, and A. Morgan. *Tate module and bad reduction*. 2018. arXiv: 1809.10208.

[18] T. Dokchitser and C. Doris. "3-torsion and conductor of genus 2 curves". In: *Math. Comp.* 88.318 (2019), pp. 1913–1927. DOI: 10.1090/mcom/3387.

[19] T. Dokchitser et al. *Arithmetic of hyperelliptic curves over local fields*. 2018. arXiv: 1808.02936.

[20] T. Dokchitser et al. *Semistable types of hyperelliptic curves*. 2017. arXiv: 1704.08338.

[21] C. Doris. *ExactpAdics: A package for exact p-adic computation*. URL: https://cjdoris.github.io/ExactpAdics.

[22] C. Doris. *ExactpAdics: An exact representation of p-adic numbers*. 2018. arXiv: 1805.09794.

[23] C. Doris. *ExactpAdics2: Another package for exact p-adic computation*. URL: https://cjdoris.github.io/ExactpAdics2.

[24] C. Doris. *Genus2Conductor: A package for computing the conductor of curves of genus 2*. URL: https://cjdoris.github.io/Genus2Conductor.

[25] C. Doris. *On enumerating extensions of p-adic fields with given invariants*. 2018. arXiv: 1803.08023.

[26] C. Doris. *pAdicExtensions: A package for computing extensions of a p-adic field*. URL: https://cjdoris.github.io/pAdicExtensions.

[27] C. Doris. *pAdicGaloisGroup: A package for computing Galois groups of p-adic polynomials*. URL: https://cjdoris.github.io/pAdicGaloisGroup.

[28] C. Fieker and J. Klüners. "Computation of Galois groups of rational polynomials". In: *LMS J. Comput. Math.* 17.1 (2014), pp. 141–158. DOI: 10.1112/S1461157013000302.

[29] E. V. Flynn et al. "Empirical evidence for the Birch and Swinnerton-Dyer conjectures for modular Jacobians of genus 2 curves". In: *Math. Comp.* 70.236 (2001), pp. 1675–1697. DOI: 10.1090/S0025-5718-01-01320-5.

[30] K. Girstmair. "On the computation of resolvents and Galois groups". In: *Manuscripta Math.* 43.2-3 (1983), pp. 289–307. DOI: 10.1007/BF01165834.

[31] C. Greve. "Galoisgruppen von Eisensteinpolynomen über p-Adischen Körpern". PhD thesis. Universität Paderborn, 2010.

[32] C. Greve and S. Pauli. "Ramification polygons, splitting fields, and Galois groups of Eisenstein polynomials". In: *Int. J. Number Theory* 8.6 (2012), pp. 1401–1424. DOI: 10.1142/S1793042112500832.

[33] A. Grothendieck and M. Raynaud. "Modèles de Néron et monodromie". In: *Groupes de Monodromie en Géométrie Algébrique*. Ed. by P. Deligne et al. Vol. 288. Lecture notes in mathematics. Springer, Berlin, Heidelberg, 1972, pp. 313–523. DOI: 10.1007/BFb0068688.

[34] W. Hart, F. Johansson, and S. Pancratz. *FLINT: Fast Library for Number Theory*. URL: http://flintlib.org.

[35] C. Helou. *Non-Galois ramification theory of local fields*. Vol. 64. Algebra Berichte [Algebra Reports]. Verlag Reinhard Fischer, Munich, 1990, p. 21.

[36] K. Hensel. "Über eine neue Begründung der Theorie der algebraischen Zahlen". In: *Jahresbericht der Deutschen Mathematiker-Vereinigung* 6 (1897), pp. 83–88. URL: http://eudml.org/doc/144593.

[37] M. van Hoeij, J. Klüners, and A. Novocin. "Generating subfields". In: *J. Symbolic Comput.* 52 (2013), pp. 17–34. DOI: 10.1016/j.jsc.2012.05.010.

[38] J. van der Hoeven, G. Lecerf, and B. Mourrain. *The Mathemagix computer algebra and analysis system*. URL: http://www.mathemagix.org.

[39] I. Itenberg, G. Mikhalkin, and E. Shustin. *Tropical algebraic geometry*. Second. Vol. 35. Oberwolfach Seminars. Birkhäuser Verlag, Basel, 2009, pp. x+104. DOI: 10.1007/978-3-0346-0048-4.

[40] J. W. Jones and D. P. Roberts. "A database of local fields". In: *J. Symbolic Comput.* 41.1 (2006), pp. 80–97. DOI: 10.1016/j.jsc.2005.09.003.

[41] F.-V. Kuhlmann. "Maps on ultrametric spaces, Hensel's lemma, and differential equations over valued fields". In: *Comm. Algebra* 39.5 (2011), pp. 1730–1776. DOI: 10.1080/00927871003789157.

[42] C. Lehr and M. Matignon. "Wild monodromy and automorphisms of curves". In: *Duke Math. J.* 135.3 (2006), pp. 569–586. DOI: 10.1215/S0012-7094-06-13535-4.

[43] Q. Liu. "Courbes stables de genre 2 et leur schéma de modules". In: *Math. Ann.* 295.2 (1993), pp. 201–222. DOI: 10.1007/BF01444884.

[44] Q. Liu. "Modèles minimaux des courbes de genre deux". In: *J. Reine Angew. Math.* 453 (1994), pp. 137–164. DOI: 10.1515/crll.1994.453.137.

[45] The LMFDB Collaboration. *The L-functions and Modular Forms Database*. URL: http://www.lmfdb.org.

[46] J. Milstead. "Computing Galois groups of Eisenstein polynomials over p-adic fields". PhD thesis. University of North Carolina at Greensboro, 2017.

[47] M. Monge. "A family of Eisenstein polynomials generating totally ramified extensions, identification of extensions and construction of class fields". In: *Int. J. Number Theory* 10.7 (2014), pp. 1699–1727. DOI: 10.1142/S1793042114500511.

[48] M. Monge. "Determination of the number of isomorphism classes of extensions of a $\mathfrak{p}$-adic field". In: *J. Number Theory* 131.8 (2011), pp. 1429–1434. DOI: 10.1016/j.jnt.2011.02.004.

[49] Y. Namikawa and K. Ueno. "The complete classification of fibres in pencils of curves of genus two". In: *Manuscripta Math.* 9 (1973), pp. 143–186. DOI: 10.1007/BF01297652.

[50] A. P. Ogg. "Elliptic curves and wild ramification". In: *Amer. J. Math.* 89 (1967), pp. 1–21. DOI: 10.2307/2373092.

[51] O. Ore. "Bemerkungen zur Theorie der Differente". In: *Math. Z.* 25.1 (1926), pp. 1–8. DOI: 10.1007/BF01283823.

[52] S. Pauli and X.-F. Roblot. "On the computation of all extensions of a p-adic field of a given degree". In: *Math. Comp.* 70.236 (2001), pp. 1641–1659. DOI: 10.1090/S0025-5718-01-01306-0.

[53] S. Pauli and B. Sinclair. "Enumerating extensions of (π)-adic fields with given invariants". In: *Int. J. Number Theory* 13.8 (2017), pp. 2007–2038. DOI: 10.1142/S1793042117501081.

[54] I. Reiner. "Integral representations of cyclic groups of prime order". In: *Proc. Amer. Math. Soc.* 8 (1957), pp. 142–146. DOI: 10.2307/2032829.

[55] J. M. Rojas and X. Wang. "Counting affine roots of polynomial systems via pointed Newton polytopes". In: *J. Complexity* 12.2 (1996), pp. 116–133. DOI: 10.1006/jcom.1996.0009.

[56] S. Rudzinski. "Symbolic computation of resolvents". MA thesis. University of North Carolina at Greensboro, 2017.

[57] The Sage Developers. *SageMath, the Sage Mathematics Software System.* URL: http://www.sagemath.org.

[58] T. Saito. “Conductor, discriminant, and the Noether formula of arithmetic surfaces”. In: *Duke Math. J.* 57.1 (1988), pp. 151–173. DOI: 10.1215/S0012-7094-88-05706-7.

[59] C. Schembri. “Modular abelian surfaces over imaginary quadratic fields”. In preparation.

[60] J.-P. Serre. *Local fields*. Vol. 67. Graduate Texts in Mathematics. Translated from the French by Marvin Jay Greenberg. Springer-Verlag, New York-Berlin, 1979, pp. viii+241.

[61] J.-P. Serre and J. Tate. “Good reduction of abelian varieties”. In: *Ann. of Math. (2)* 88 (1968), pp. 492–517. DOI: 10.2307/1970722.

[62] B. Sinclair. “Algorithms for enumerating invariants and extensions of local fields”. PhD thesis. University of North Carolina at Greensboro, 2015.

[63] B. Sinclair. *Counting Extensions of $\mathfrak{p}$-Adic Fields with Given Invariants*. 2015. arXiv: 1512.06946.

[64] B. Sinclair. *Number of extensions of local fields*. URL: http://www.uncg.edu/mat/numbertheory/tables/local/counting.

[65] R. P. Stauduhar. “The determination of Galois groups”. In: *Math. Comp.* 27 (1973), pp. 981–996. DOI: 10.2307/2005536.

[66] J. Tate. “Algorithm for determining the type of a singular fiber in an elliptic pencil”. In: *Modular Functions of One Variable IV*. Vol. 476. Lect. Notes in Math. Springer-Verlag, Berlin, 1975, pp. 33–52. DOI: 10.1007/BFb0097580.

[67] D. Ulmer. “Conductors of ℓ-adic representations”. In: *Proc. Amer. Math. Soc.* 144.6 (2016), pp. 2291–2299. DOI: 10.1090/proc/12880.